FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Development of an Educational Software Tool for the Topic of Group Communication

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Nicholas Alexander Stifter
Matrikelnummer 0306914

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Mag.rer.soc.oec. Dipl.-Ing. Dipl.-Ing. Dr.techn. Karl Michael Göschka
Mitwirkung: Mag.rer.soc.oec. Dipl.-Ing. Dr.techn Lorenz Froihofer

Wien, 26.11.2015 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuer)

DIPLOMA THESIS

Development of an Educational Software Tool for the Topic of Group Communication

Submitted at the
Faculty of Informatics,
Vienna University of Technology
in partial fulfilment of the requirements for the degree of

Magister der Sozial- und Wirtschaftswissenschaften

under supervision of

Priv.-Doz. Mag.rer.soc.oec. Dipl.-Ing. Dipl.-Ing. Dr.techn.
Karl Michael Göschka

Institute number: 184-1
Institute of Information Systems
Distributed Systems Group

by

Nicholas Alexander Stifter, Bakk. techn.
Matr.Nr. 0306914
Eisenstädterstraße 8
7350, Oberpullendorf

26. November 2015

Erklärung zur Verfassung der Arbeit

Nicholas Alexander Stifter, Eisenstäderstraße 8, 7350 Oberpullendorf

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 26. November 2015,

Dedication

I would like to dedicate this work to my partner Natalie Denk.

Without your support and love it would not have been possible.

Abstract

This thesis covers the development of an educational software framework aimed at assisting learners in the implementation and testing of reliable multicast protocols and other software components related to (process) group communication. The motivation for this work stems from the intention of extending an existing group communication simulator that is used in an advanced distributed systems course held at the Vienna University of Technology. Currently student-developed protocols are restricted to the simulation environment and cannot be used in real-world scenarios. Ideally students should be able to develop and test their own protocol implementations using tools that foster the learning process and simplify the complex task of creating reliable distributed software while retaining its full functionality. Interestingly, at the time of writing there was very little available literature covering the design and development of educational software tools for more advanced distributed computing topics and group communication in particular.

Part one of this work therefore provides an overview of relevant literature for developing educational software tools for the topic of group communication. The term *educational group communication system* or *eGCS* is introduced to conceptualize a software framework that combines all the desirable attributes of such an educational tool. It is highlighted that further research on the subject matter is needed and the presented results should be seen as an initial step towards formulating guidelines for creating *e*GCSs or other educational tools for more advanced distributed systems topics.

Part two covers the development of a prototype *eGCS* aimed at solving the initial problem of extending a group communication simulator and is informed by the results presented in part one of this thesis. The proposed *e*GCS design employs a *header-driven* model for protocol composition, an extended variant of Chandra-Toueg Consensus as the basis for a combined group membership service and view synchronous multicast communication protocol, and the distributed middleware simulation environment *MINHA* to address the issue of allowing protocols developed for the simulation environment to be usable in real-world scenarios. MINHA virtualizes the execution of multiple Java virtual machine (JVM) instances in a single JVM, where key components such as network sockets and threads are replaced with simulated counterparts through bytecode instrumentation. Rather than relying on a group communication simulator for protocol development, regular protocol stacks are used and can be transformed through MINHA to execute in a simulated environment when needed. A preliminary implementation of the proposed *e*GCS outlines the design's feasibility, however seldom but hard to debug issues that can potentially be introduced through MINHA's bytecode instrumentation process render the suitability of the chosen simulation approach questionable for an educational scenario at this point in time.

Kurzfassung

Die vorliegende Arbeit befasst sich mit der Entwicklung eines didaktischen Softwareframeworks für die Implementierung und Testung zuverlässiger Multicastprotokolle und anderen Softwarekomponenten aus dem Themenbereich Gruppenkommunikation (*group communication*). Die Motivation dieser Arbeit entspringt dem Vorhaben der Erweiterung eines existierenden Simulators für Gruppenkommunikation. Dieser wird in der Laborübung einer vertiefenden Lehrveranstaltung zum Thema Verteilte Systeme an der TU Wien eingesetzt. Die im Rahmen des ursprünglichen Simulators von Studenten entwickelten Multicastprotokolle sind an die derzeitige Simulationsumgebung gebunden und können nicht ohne weitläufige Modifikationen in realistischen Einsatzszenarien verwendet werden. Ein derartiges didaktisches Software-Framework sollte jedoch idealerweise nicht nur den Lernprozess bei der Entwicklung unterstützen und fördern, sondern auch Lernenden die Möglichkeit bieten voll funktionsfähige, zuverlässige verteilte Software zu implementieren.

Literatur, die Vorgehensweisen oder Systemarchitekturen für die Entwicklung einer didaktischen Software für Gruppenkommunikation oder fortgeschrittene Themen aus dem Bereich Verteilte Systeme beschreibt, war zum Zeitpunkt der Erstellung kaum vorhanden. So beschäftigt sich der erste Teil der Arbeit mit dem aktuellen Stand der Forschung zum Thema und bietet eine Übersicht relevanter Literatur für die Entwicklung didaktischer Software-Frameworks für Gruppenkommunikation. Der Begriff *educational group communication system* oder *eGCS* wird eingeführt und soll ein konzeptionelles Framework beschreiben, das alle wünschenswerten Eigenschaften in sich vereint. Die Ergebnisse des ersten Teils der Arbeit sind als ein initialer Schritt in Richtung einer Formulierung für Richtlinien zur Erstellung solcher didaktischen Software-Frameworks zu verstehen und veranschaulichen den Bedarf einer weitergehenden wissenschaftlichen Auseinandersetzung mit der Thematik.

Teil zwei befasst sich mit der Entwicklung eines *e*GCS Prototypen zur Lösung des anfänglich genannten Vorhabens der Erweiterung eines Gruppenkommunikationssimulators. Der Designvorschlag basiert auf einem Kompositionsmodell von Protokollen nach dem *header-driven* Ansatz, einer erweiterten Implementierung des Chandra-Toueg Konsensus Algorithmus, der als Basis eines kombinierten Protokolls für Gruppenmitgliedschaft (*group membership*) und virtuell synchrone Multicast-Kommunikation (*view synchronous multicast communication*) dient, sowie der Simulationsumgebung MINHA zur Simulation und Testung von Gruppenkommunikations-protokollstacks. MINHA ermöglicht eine virtualisierte Ausführung multipler Java virtueller Maschinen (JVM) in einer einzigen JVM, in der Kernelemente wie Netzwerk-Sockets und Threads durch simulierte Komponenten ersetzt werden. Hierzu wird *Bytecode Instrumentation* verwendet, die regulären Java Bytecode so transformiert, dass dieser zu der unterliegenden ereignisorientierten Simulation synchronisiert ausgeführt wird. Es ist demnach möglich regulären Code zu entwickeln und diesen bei Bedarf ohne weitere Modifikationen mittels MINHA als Simulation auszuführen. Eine vorläufige Implementierung des *e*GCS Designvorschlags zeigt dessen Umsetzbarkeit, jedoch stellen selten auftretende, aber durch den Prozess des Bytecode Instrumentierens schwer zu lokalisierende Fehler die Verwendung MINHAs in einem pädagogischen Kontext momentan in Frage.

Table of contents

Abbreviations and Symbols

ADS	Advanced Distributed Systems
API	Application Programming Interface
AV	Algorithm Visualization
DI	Dependency Injection
FIFO	First In First Out
*e*GCS	educational Group Communication System
GCS	Group Communication System
GMS	Group Membership Service
GUI	Graphical User Interface
IDE	Integrated Development Environment
IoC	Inversion of Control
JVM	Java Virtual Machine
PBL	Problem Based Learning
PCGM	Primary Component Group Membership
RMI	Remote Method Invocation
SDL	Specification Description Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VSC	View Synchronous Communication
VSGMS	View Synchronous Group Membership Service
WAN	Wide Area Network

$\mathcal{AP}$	Almost Perfect Failure Detector
$\mathcal{P}$	Perfect Failure Detector
$\diamond\mathcal{P}$	Eventually Perfect Failure Detector
$\mathcal{S}$	Strong Failure Detector
$\diamond\mathcal{S}$	Eventually Strong Failure Detector
$\diamond\mathcal{W}$	Eventually Weak Failure Detector

1. Overview

1.1 Introduction

Developing and implementing distributed algorithms, protocols, and distributed software in general can be a difficult and error prone task, even if the developer is well versed in the topic area. This difficulty might be attributed to different aspects of distributed systems that can come into play, such as the lack of a global clock, unreliable communication channels, and the potential failure of processes. From a practical standpoint the formal concepts offered by the research community are often not trivially translated into concrete software applications. Great care needs to be taken when designing and developing such an implementation as to not invalidate any formal guarantees by adapting or modifying the underlying algorithm to fit a specific use case. With sufficient complexity it can also become prohibitive to provide a formal proof of the correctness for a particular software composition. Components that in themselves are correct may be erroneously composed because their functionality or provided guarantees were not properly understood by the composer. Such mistakes are often difficult to pinpoint because the resulting program is actually behaving correctly according to its erroneous design.

The focus of this thesis lies within the topic area of *group communication* and is centred on *reliable multicast protocols* which are used to send messages to the members of a *group*, and that provide certain guarantees about the delivery and ordering of messages. The extension of an existing laboratory assignment that is part of a university course on advanced distributed systems topics serves as the basis and motivation for this work ([GF11],[FG11]). In this laboratory students are tasked with implementing their own reliable multicast protocols as well as extending a simple, turn-based game to execute in a fault tolerant, distributed manner. Currently both of these tasks form separate problems because students need to develop their own protocol implementations within a provided group communication simulator and cannot use them for later parts of the assignment. Ideally students should be able to rely on their self-developed protocols for this second task and not have to draw upon external solutions. This thesis addresses the issue of how such an endeavour might be achieved. In particular the question is raised how students can be assisted in the development and testing of such multicast protocols and other components of a *group communication system*[1] (GCS) in light of the aforementioned difficulties encountered when developing distributed software.

[1]In this work group communication system and group communication service are used synonymously. Works such as [CKV01] also make no clear distinction of the terms, however a group communication system may be considered a concrete implementation while group communication service refers to the services it provides.

1.2 Problem Statement

Group communication is an important topic area of distributed computing. It is closely related to fundamental concepts such as *Consensus* and *failure detection* and highlights essential properties in distributed systems such as being able to provide the guarantee that all members of a group receive the same set of messages in the wake of potential failures of both processes and the channels through which they communicate. While the abstractions provided by group communication primitives are often used in the context of *active replication*, they can serve a multitude of purposes in distributed systems. There exist various tools and frameworks for developing software related to group communication. The overwhelming majority however is targeted towards an application in commercial environments or for research purposes. If we shift the focus towards computing education it becomes apparent that at the time of writing there is a distinct lack of widely published works covering the design and development of educational tools for advanced distributed computing topics. The author is unaware of a software tool specifically aimed at assisting in the development of components related to group communication systems such as reliable multicast protocols in an educational context[2]. Educational tools for distributed computing in general seem rather scarce in comparison to the abundance of works that can be found for introductory topics, such as learning a programming language or understanding basic algorithms. This observation is not just limited to distributed computing, but can be said of many of the more advanced topic areas in computer science. A distinction has to be made between educational tools aimed at teaching (abstract) computer science concepts and those that assist the learner applying these concepts to concrete, real world examples. Works in the former group generally do not need to offer students the ability to implement fully functional code and focus on effectively conveying the presented concepts to learners. The latter class of tools on the other hand is geared at simplifying or assisting the (software) development process and offers learners the ability to apply and strengthen their knowledge through practical coursework. This thesis deals with the development of an educational tool or framework for the creation of software components centred around group communication and hence falls into the second category of works.

Unfortunately there also exist no clear guidelines on the design of educational software tools for advanced distributed systems topics. The focus is often cast on a less experienced audience and introductory courses where flexibility and functionality of the framework can more readily be given up to include such aspects as visualization or the ability to dynamically interact and modify the execution steps of distributed algorithms (such as [Ben-A01]). Protocol composition frameworks and modular GCSs can aid researchers and protocol developers in the implementation and testing of protocols and services in the context of group communication. They are however generally not geared towards an application in an educational context and can be complex and cumbersome to use. In this regard an important question that needs to be raised is if an educational tool for developing group communication components could offer significant learning advantages over “regular” tools that are also designed for that task. Without any previous literature on developing and designing such an educational tool, a tool we will refer to as an *educational Group Communication System or eGCS*, a broad overview of the general topic area for both educational and non-educational works is

[2]Beyond the group communication simulation environment of the advanced distributed systems laboratory [GF11] that serves as a basis for this thesis.

required to obtain an overview of the state-of-the-art and be able to make informed design decisions. It has been mentioned in the Introduction that the extension of an existing laboratory assignment on advanced distributed systems topics serves as the basis for this work. This specific problem is covered in part two of this work (see “Development of a Prototype Educational Group Communication System“) while the more generalized question of:

> How should an educational tool be designed to assist in both the creation and testing as well as the understanding of (reliable) multicast protocols and other components related to group communication while also allowing learners to use these components in regular, real-world scenarios?

is addressed in part one (“Educational Software Tools for the Topic of Group Communication“).

1.3 Scope

The scope of this thesis is both the design of an *e*GCS prototype as an extension of an existing group communication simulator used in an advanced distributed systems laboratory as well as the formulation of a general design approach for developing educational (software) tools for advanced distributed systems topics and group communication in particular. Literature on both protocol composition frameworks for group communication (and modular GCSs) and educational software frameworks for the implementation of distributed algorithms and protocols is to be surveyed and their suitability for this particular usage scenario discussed. This discussion is framed by an overview of the fundamental concepts of group communication and the development and testing of group communication systems. To the author's knowledge there exists no educational software framework directly focused on the topic of group communication. As such more general educational software frameworks for distributed algorithms have to be included and their design considerations need to be compared and weighed against those of “regular” non-educational protocol composition frameworks and modular GCSs. The lack of previous works covering the development and testing of group communication components in an educational context clearly warrants a more in-depth discussion on the topic area.

One explicit goal of this thesis is to allow students to use their own multicast protocol implementations for the second part of the laboratory assignment. The current simulation model provides asynchronous reliable point to point communication channels and uses *static group* semantics while the second part of the laboratory allows for *dynamic group membership* and reliable multicast communication through the GCS middleware *Spread*[3] ([AS98]). Dynamic group semantics and *virtual synchrony* will have to be introduced to the simulation environment so students can also rely on these features. Part two of this thesis deals with the development of an *e*GCS prototype that addresses these issues and is specifically tailored towards the requirements of the advanced distributed systems laboratory assignment.

[3] Spread provides partitionable group membership and is based on the concept of *extended virtual synchrony* (see [MAMA94]).

1.4 Document Structure

This thesis is split into two distinct parts. Part One (2) covers the theoretical aspects of developing an educational group communication system. Fundamental concepts in regards to group communication are presented to provide the uninformed reader with a starting point for further research and serve as a basis for discussion (2.2). Essential aspects such as Consensus, failure detection, group membership and virtual synchrony are outlined in more detail. The reader is then presented with the topic of protocol composition frameworks and group communication systems to illustrate different approaches on how modern, modular group communication protocol stacks are composed (2.3). In chapter (2.4) the simulation and testing of GCSs and more complex protocol stacks is discussed. Being able to test and simulate the execution of distributed software and in particular group communication systems is an essential capability and an integral component of an *e*GCS. Several different approaches are outlined ranging from abstract simulations of view synchronous communication to methods such as using bytecode instrumentation to transform regular GCS code so it runs in a simulated environment. The focus is then shifted towards educational tools used for teaching distributed systems concepts. As has been outlined in the Introduction (1.1) and Problem Statement (1.2) at the time of this writing no publications were available that cover the development of educational tools for the topic of group communication. Hence the broader topic area of distributed algorithms is considered in chapter (2.5), covering works that present educational tools for distributed algorithms and protocols aimed at more introductory level. Part one is then concluded with a general discussion on the development of an educational group communication system, outlining that further research in the topic area is needed (2.6).

Part Two (3) deals with the development of an eGCS prototype aimed at extending the current group communication simulator. In chapter (3.2) an outline of the current laboratory design is given and the primary goals of rendering student protocols usable outside of the simulation environment as well as the extension of the current simulator to include dynamic group semantics and virtual synchrony are defined. Chapter (3.3) then discusses these goals in the context of the results presented in part one and how they may influence any design decisions. A design proposal for an eGCS is made in (3.4) that is based on a *header-driven* protocol composition model and uses *Consensus* as a basic building block for implementing *primary component group membership* and virtual synchrony. Simulation and testing facilities are provided through the distributed middleware simulation environment MINHA, that employs bytecode instrumentation to transform regular Java code so it executes as a discrete event simulation. It is argued that such a new design may be better suited for an educational application than extending an existing GCS or protocol composition framework. (3.5 Prototype Development) outlines the implementation of a prototype based on the proposed design to test its feasibility. Initial results are promising, however MINHA can produce hard to debug issues under certain circumstances that render its application in an educational setting questionable at this point in time. Chapter (3.6) then gives a brief overview of the outcome of part two.

Finally, in (4) the Conclusion of this work is presented where the results of part one and two are discussed and future work on the topic area is outlined.

Part One

2. Educational Software Tools for the Topic of Group Communication

2.1 Overview

This part of the thesis covers the design and development of educational tools for the topic area of group communication. Specifically, it deals with the following question put forth in the initial problem statement:

“How should an educational tool be designed to assist in both the creation and testing as well as the understanding of reliable multicast protocols and other components related to (view synchronous) group communication while also allowing learners to use these components in regular, real-world scenarios?”

Considering that research around group communication has been ongoing for well over two decades it is surprising how hard it is to find resources on educational group communication tools or even teaching group communication concepts in general. One might speculate that, as a rather advanced topic, the target learner audience is mostly composed of senior students or researchers interested in that particular field to whom scientific publications and technical manuals are common learning materials. At closer observation however a lot of research in the field is focused towards modular and readily extensible group communication middleware, easier and more flexible protocol composition, and better testing and simulation facilities. This may be an indicator that also in the scientific community the need for more straightforward and easier to use tools is present.

Regardless of the particular reasons for a lack of publications on educational group communication tools, their design, governing pedagogy, and expected educational effectiveness in regards to regular tools needs to be discussed in the context of this thesis. The focus of this work is not on devising general guidelines for building such educational tools, yet without any directly related literature on the topic area at least an initial step in this direction has to be taken. In the following chapters the reader is introduced to the fundamental concepts and aspects of view synchronous group communication. It includes topics such as *agreement, failure detection, group membership, and multicast communication.* For *group communication systems (GCSs)* generalized modular specifications, their composition using *(protocol) composition frameworks*, as well as their simulation and testing is covered. To compensate for the lack of works covering the topic area, educational tools from the more general field of distributed computing that focus on distributed algorithms and protocols are included. These tools are mostly aimed at less experienced learners and it is unclear if the insights and findings can be directly applied to the considered target audience. General approaches for designing coursework for advanced distributed computing topics are also mentioned. Finally, possible design considerations for an educational tool for developing group communication components, a tool we shall refer to as the *educational group communication system* or *eGCS* are discussed. An *eGCS* should be understood as a unifying descriptor for composition frameworks, modular GCS protocol stacks and simulation and testing environments that are used together in an educational context.

In summary part one of this work covers the development of guidelines for creating educational group communication systems and educational tools related to group communication from different perspectives. It also highlights the need for more publications on the topic of creating educational tools for more advanced computer science topics, particularly in the area of distributed computing.

2.2 Fundamental Concepts

2.2.1 Considered Topic Area: Group Communication

Group communication paradigms can be difficult to convey to learners not familiar with fundamental concepts of distributed systems. Loosely speaking, group communication addresses many of the problems faced when attempting to reliably send messages to more than one recipient. What might appear to the uninformed reader as a straightforward issue is actually very faceted, drawing upon many fundamental problems encountered in distributed systems. Consider the premise that a *protocol* has to deliver a message to all correct recipients - if that message has been delivered to any recipient, faulty or not. A protocol that provides such a guarantee is referred to as being *uniform.* How are the recipients of the message defined? They could be a predefined static set of processes or dynamically change over time. If this set can change how can we ensure that processes *agree* on the same set of processes that make up the group at any one time. Additionally, we face the issue of potential failures of both processes and the communication channels they use. Without specifying time bounds for process execution speeds and message transmission times it even is impossible for a process to deterministically decide if another process has failed or not ([FLP85]). From this initial premise it becomes clear that a meaningful discussion on the topic of group communication requires that many other aspects of distributed systems also need to be addressed and clarified.

This chapter is set out to provide an overview of the essential concepts behind group communication, the general topic area of this thesis. First basic definitions and assumptions of the system model are presented to form a basis for the discussion of more advanced concepts. Next an overview of failure detection is given followed by the topic of *agreement*. The focus is then placed on *multicast protocols* with different guarantees that are used for group communication. Finally both the *group membership problem* and *virtual synchrony* are covered.

2.2.2 System Model and Definitions

The assumed system model greatly affects how and even if a particular problem may be solvable and also serves as a basis for discussions. As such it should be carefully chosen to match the expected conditions while not being so restrictive as to only apply to a very specific scenario or present itself as too impractical to be implemented in real world applications. This chapter gives a basic overview of some of the essential components of a distributed system such as *processes* and the *channels* connecting them as well as describing different possible *failure modes* and *synchrony assumptions* for these components.

Distributed System

The following quote from a textbook on distributed systems by Tanenbaum and Steen probably best describes the situation when trying to define what constitutes a distributed system:

> “Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others.” ([TS07])

Their definition of a distributed system loosely characterizes it as:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system." ([TS07])

Other definitions of distributed systems are given, for instance, by Coulouris et al:

> "We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages." ([CDK+11])

or Gosh, who lists the criteria as multiple processes with disjoint address spaces using interprocess communication to meet a collective goal ([Gho07]). A similarly generalized definition for what Birman refers to as a distributed computing system[4] is:

> "Reduced to the simplest terms, a *distributed computing system* is a set of computer programs, executing on one or more computers, and coordinating actions by exchanging *messages*." ([Bir05])

To this extent Birman goes as far as including distributed systems based on a single multitasking computer and systems in which information flow between components occurs by means other than message passing such as using shared memory. These examples serve to illustrate the difficulty in creating a brief unifying description of what constitutes a distributed system. Rather than attempting to also define the considered distributed system in a generalized manner, we will describe individual aspects and properties. When specifying problems or distributed algorithms certain assumptions about the underlying system model need to be made to form a common basis for discussion. The important characteristics to consider, also in respect to group communication in distributed systems, are both the *synchrony* and *failure model* of the system ([Men06]). First, however we need to briefly cover the basic components that make up the distributed system, namely *processes* and their underlying method of communication.

Processes

Tannenbaum and Steen describe processes as:

"The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. " ([TS07])

Similar definitions can be found in ([Bir05])

"a program is a set of instructions, and a process denotes the execution of those instructions."

and ([CGR11])

"We abstract the units that are able to perform computations in a distributed system through the notion of a process."

When talking about processes the abstract notion of some unit executing a program is used rather than going into specifics on the actual hardware or how that process is implemented in the operating system.

[4]While Tanenbaum & Steen refer to distributed computing systems as a subgroup of distributed systems focussed on high performance computing tasks [TS07] Birman's conception of the term roughly equates to general definitions of distributed systems [Bir05]

Connectivity

In a distributed system processes should be able to exchange information with one-another. Communication may be conducted through message passing or through other means such as using distributed shared memory. The message passing model is a commonly used approach and is discussed here in more detail. Messages are assumed to be passed from one process to another through a so called *channel* or *link.* For formal specifications channels are often modelled as directional or bi-directional edges in a graph where processes form the vertices. If a pair of processes is connected through a bidirectional channel that is exclusively reserved for their communication this is called *point-to-point links*. It is often assumed that there exists such a point-to-point link for any two processes p and q, forming a complete graph of bidirectional links between processes. Communication links or channels can be qualified on the basis of their properties, such as: ([Men06])

1. No-Creation. If process *q* receives message *m* from *p*, then *p* has sent *m* to *q*.
2. No-Duplication. If process *q* receives message *m* from *p*, *q* receives *m* from *p* at most once.
3. No-Loss. If process *p* sends message *m* to *q*, and both processes are correct, then *q* eventually receives *m* from *p*.
4. Fair-Loss. If process *p* sends an infinite number of messages to *q* and *q* is correct, then *q* receives an infinite number of messages from *p*.

Channels are called *quasi-reliable* if properties 1, 2, and 3 hold. The stronger definition of *reliable channels* assumes for property 3 (No-Loss) that only the receiver must be correct. Reliable channels are not practical to implement as they assume that the channel does not lose any messages[5] ([GS01]). With an *eventually reliable channel* there is a time after which all messages sent are eventually received, however there exists a finite but unbounded number of messages that may be lost ([BCT96]). While many problems such as r*eliable-*, *FIFO (First In First Out)-*, and c*ausal broadcast* are solvable using eventually reliable channels, some problems such as u*niform reliable broadcast* are not (see [BCT96]) and require further assumptions.

Naturally these properties do not reflect the reality of actual point-to-point connections that can fail and lose messages. Through buffering and employing re-transmission schemes as well as having processes relay messages many of the failures of links can be masked. Nevertheless real connections follow a *best effort* approach and may not be able to uphold the specified guarantee for every execution. The above specifications however are still useful because in the majority of cases the network will “behave”. If the underlying network permanently fails to meet any reasonable (*liveness)* guarantees it is clear that no sensible discussion can be held, as it becomes impossible for processes to communicate. As such specifications often assume that the network is *stable* or has long enough *stable periods* for the distributed algorithm to make progress.

One important aspect to consider is the possibility of network partitions to occur. In a partition a subset of the processes is isolated from the rest, meaning not a single process of the partition shares a channel to a process that is not part of the partition. The observation of partitions is intimately linked to timing assumptions and failure detection. If communication channels are *asynchronous* and

[5]Since a reliable channel must deliver messages also sent by faulty processes, if such a message is lost during transit and the sending process has failed there may be no way to recover the message and hence the guarantee cannot be met.

their failure is not accurately detectable it is impossible to deterministically determine if a partition has occurred as one would have to wait indefinitely to be certain that messages from the suspected partition are not just very late.[6] If we assume that channels do not fail permanently, processes fail in the *crash-stop* model, and the network stabilizes, eventually a single *stable component*[7] will form. On the other hand if channels can fail permanently several partitions may exist that all form stable components. Chockler et al. define the stable component as:

> "A stable component is a set of processes that are eventually alive and connected to each other and for which all the channels to them from all other processes (that are not in the stable component) are down." ([CKV01])

Synchrony

There are several different timing assumptions that can be made for processes and the channels connecting them that directly relate to the synchrony of the system model. Here the properties as presented in ([Déf00]) are used. For processes the possible properties regarding synchrony and timing apply to their *relative speed* while for channels the *communication delay* for message transmission is considered. These are:

1. There is a known upper bound which always holds
2. There is an unknown upper bound which always holds.
3. There is a known upper bound which eventually holds forever.
4. There is an unknown upper bound which eventually holds forever.[8]
5. There is no bound on the value of the parameter.

On the one extreme is a system model where property 1 applies to both process speeds and communication channels. Such a system is considered to be *synchronous*. On the other is a system in which neither relative process speed nor message transmission times are bounded (property 5). This is referred to as an *asynchronous* system. In a synchronous system model many problems such as distributed Consensus are more readily solvable ([FLP85,Fuzz08]), however the required guarantees may prove to be too restrictive for a given practical implementation. In a purely asynchronous system there are no such restrictive timing assumptions but this model has been shown to be too weak for some problems to be solvable, such as deterministically reaching distributed Consensus even if just a single process can fail ([FLP85], often referred to as the *FLP impossibility result*). Between both of these extremes lie several *partially synchronous* system models with varying degrees of synchrony, in some of which Consensus can be solved ([DDS87,DLS88]). Being able to reach Consensus in the used system model is important not just because it is a basic building block of group communication systems, but also because (process) group membership is also a particular type of agreement problem. Chandra and Toueg have shown that distributed Consensus is solvable in an asynchronous system augmented with *unreliable failure detectors* ([CT96]). This however does not circumvent the FLP impossibility result and the failure detectors themselves still require a model

[6] If message transmission time is unbounded and channels are fail-silent (no more messages are transmitted) it becomes impossible to discern between a failed channel and one where messages are merely taking a long time. The FLP impossibility result [FLP85] to which this is related is discussed in the subsection on synchrony and the agreement section.

[7] Under the assumption that processes are connected through a complete graph of channels.

[8] There exist many other possible assumptions, such as: There is a known upper bound that holds infinitely often for periods of a known duration. Also see [DLS88] for a discussion on *Global Stabilization Time*.

of partial synchrony. Such an approach however allows problems to be expressed without any synchrony assumptions but rather through the *weakest class of failure detector* required to solve them (A more detailed description on failure detectors can be found in section 2.2.3). In regards to actual implementations the assumption of an asynchronous system model (augmented with failure detectors) can provide advantages. Algorithms do not have to make timing assumptions and are hence more readily adaptable to run on different systems. Furthermore it is easy to envision an unexpected scenario in which the assumed bounds of a synchronous system do not hold, rendering these synchrony assumptions at best probabilistic ([CT96]).

Failure Models

In respect to the presented model failures can manifest themselves both at processes or the communication channels with which they interact. As the model dictates that processes rely purely on message passing to communicate it may be impossible for a process to discern between a failure of the communication channel or the respective process to which that channel leads.

([Déf00]) describes the following classes of process failures:

1. *Crash failures*. When a process crashes, it ceases functioning forever. This means that it stops performing any activity including sending, transmitting, or receiving any message.
2. *Omission failures*. When a process fails by omission, it omits performing some actions such as sending or receiving a message.
3. *Timing failures.* A timing failure occurs when a process violates one of the synchrony assumptions. This type of failure is irrelevant in asynchronous systems.
4. *Byzantine failures*. A Byzantine component is allowed any arbitrary behavior. For instance, a faulty process may change the content of messages, duplicate messages, send unsolicited messages, or even maliciously try to break down the whole system.

We distinguish between *correct* and *faulty* components. A *correct* component will *never* exhibit any faulty behaviour during the lifetime of the system while components are considered to be *faulty* even if they have not yet shown any *incorrect* behaviour.

Assumed System Model

The following system model is generally assumed for presented algorithms and concepts unless otherwise specified: The distributed system is modelled as a finite set of processes $\Pi = \{p_1, p_2, p_3 \ldots, p_n\}$ that are completely connected by bidirectional point-to-point channels. Communication between processes is restricted to messages that can be passed on channels. The system is asynchronous in both process execution and message transmission time but is augmented with unreliable failure detectors[9]. Faulty processes fail in the crash-stop model and do not exhibit Byzantine behaviour. Channels are assumed to be quasi-reliable. Furthermore the network is assumed to be stable for long enough periods to allow distributed algorithms to make progress.

[9]Without this augmentation the [FLP85] impossibility result holds and many problems such as Consensus become unsolvable.

2.2.3 Failure Detection

After having introduced the system model and failure modes in section (2.2.2) the focus is now placed on how a distributed system may try to detect or otherwise deal with faulty components. Some failures such as process crashes have the peculiarity that they generally can't be avoided, but rather have to be dealt with indirectly though means such as adding redundancy to the system. One cannot expect that a faulty process is itself able notify its peers of its own failure before crashing. Generally speaking, a process will query some form of *oracle* on the status of other processes to determine if they have failed or not. As the name implies, the information an oracle provides need not be accurate or even correct. Commonly used oracles in distributed systems are *physical clocks*, *failure detectors*, and *randomization* ([Déf00]):

Physical clocks will give a process an indication of how much time has passed. This is especially relevant for synchronous system models with time bounds. The clocks of a distributed system can be synchronized in respect to each other with real-time. A process can query a physical clock and use it to decide if another process is acceptably late or has exceeded the admissible time bound[10].

A *failure detector* is a component which, when queried about the status of a specific process, will usually return a boolean value if it *suspects* that process of having failed. As stated before this information need not be correct and two different processes querying a single failure detector on the status of a particular process may receive different answers.

Aside from physical clocks and failure detectors *randomization* can also be used as an oracle. Such *coin flips* for instance also allow Consensus to be solved in an asynchronous system with a probability that converges towards one. This however should not be confused with deterministically solving agreement.

In their seminal work on *unreliable failure detectors* Chandra and Toueg consider *distributed failure detectors* that are built using unreliable local *failure detector modules*. They propose to characterize abstract classes of failure detectors through a completeness and accuracy property. *Completeness* requires that a failure detector eventually suspects every crashed process and *Accuracy* restricts the mistakes a failure detector can make. Eight different classes of failure detectors are presented with the following two *Completeness* and four *Accuracy* properties ([CT96]):

1. *Strong Completeness*. Eventually every process that crashes is permanently suspected by *every* correct process.
2. *Weak Completeness.* Eventually every process that crashes is permanently suspected by *some* correct process.
3. *Strong Accuracy*. No process is suspected before it crashes.
4. *Weak Accuracy*. Some correct process is never suspected.
5. *Eventual Strong Accuracy.* There is a time after which correct processes are not suspected by any correct process.
6. *Eventual Weak Accuracy.* There is a time after which some correct process is never suspected by any correct process

[10] A process may even need to consider itself failed if it detects that it has exceeded the admissible time-bounds.

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Fig. 1: Eight classes of failure detectors defined in terms of accuracy and completeness. [CT96]

In asynchronous systems it becomes impossible to differentiate between a crashed process or one that is just very slow ([FLP85]). Hence at least some synchrony assumptions need to be introduced if a failure detector is to be implementable. Chandra and Toueg argue that "The problem of implementing a certain type of failure detector in a specific model of partial synchrony becomes a separate issue; this separation affords greater modularity." ([CT96]). An alternative approach that does not require such synchrony assumptions is to use randomization (or *coin-flips*) to resolve this issue[11]. Problems can be characterized by the weakest class of failure detector required for solving them. *Terminating Reliable Broadcast* for instance is only solvable using a perfect failure detector in $\mathcal{P}$ and cannot be solved using either $\mathcal{S}$, $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ ([CT96]).

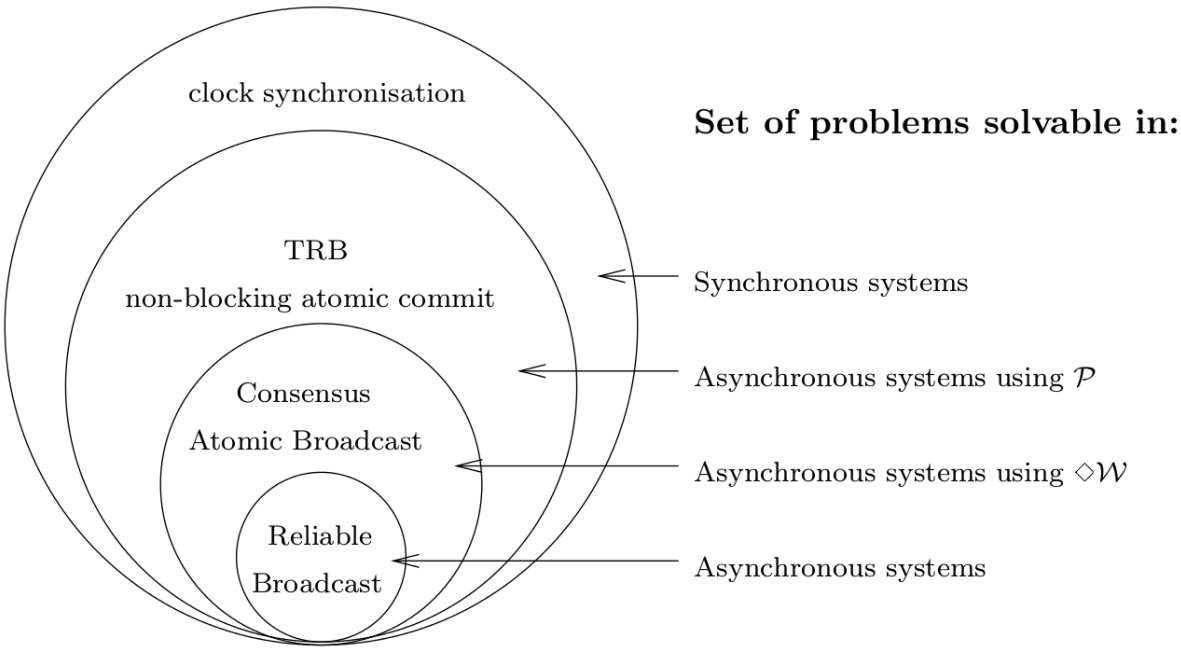

Fig. 2: Problem solvability in different distributed computing models. [CT96]

[11]See [Ben-O83] for a Consensus algorithm based on randomization.

Of particular interest is the class of $\diamond\mathcal{W}$ failure detectors which satisfies *weak completeness* and *eventual weak accuracy*. It is the weakest class of failure detectors Chandra and Toueg consider in their paper and they show that $\diamond\mathcal{W}_0$ in the class of $\diamond\mathcal{W}$ failure detectors is the weakest failure detector required for solving Consensus in an asynchronous system. More specifically, Chandra and Toueg introduce the concept of *reducibility* among failure detectors where a failure detector $\mathcal{D}'$ is reducible to failure detector $\mathcal{D}$ if there is a distributed algorithm that can transform $\mathcal{D}$ into $\mathcal{D}'$ ($\mathcal{D}$ and $\mathcal{D}'$ are considered *equivalent*). They proceed to present a distributed algorithm (processes disseminate their failure suspicions to other processes) that transforms any class of failure detector exhibiting *weak completeness* into one exhibiting *strong completeness.* This reduces the requirement to show that Consensus is solvable to failure detectors satisfying strong completeness, namely $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$. Since $\mathcal{P} \geq \mathcal{S}$ and $\diamond\mathcal{P} \geq \diamond\mathcal{S}$ it is only necessary to provide proofs for solving Consensus using both $\mathcal{S}$ and $\diamond\mathcal{S}$, both of which are presented in ([CT96]).

Implementing an (almost) perfect failure detector

As has been indicated some problems require a perfect failure detector in $\mathcal{P}$ to be solvable. This class of failure detectors generally proves to be too strong an assumption for many real systems to be implementable, however through the use of a method called *program controlled crash*[12] correct processes that were wrongfully suspected by the failure detector can be forced to conform to its incorrect assumption and must “crash” themselves. Hence the failure detector *a posteriori* never makes any mistakes and emulates a perfect failure detector. ([DSU04]) describe the following weakened accuracy property for an *almost* perfect failure detector $\mathcal{AP}$:

1. *Quasi-Strong Accuracy* No correct process is ever suspected by any correct process.

This approach was first presented by Birman et al. in the ISIS group membership system ([BJ87], [RB91]). The output (*views*) of a primary-component group membership service, if used as a mechanism for failure detection, also behaves like such an *almost perfect failure detector*[13].

There are several shortcomings when employing an almost perfect failure detector, such as weakening liveness guarantees. Fault tolerant algorithms can only tolerate the failure of a bounded number of processes ([DSU03]) and any incorrect suspicions using such an $\mathcal{AP}$ failure detector can lead to a further reduction in the resilience of the system as correct processes are forced to fail. Program controlled crash may even lead to a “collective suicide” where all processes incorrectly suspect each other. Failure detectors under such a model will need to weigh carefully if they suspect a process, which will usually be reflected by long time-out values.

2.2.4 Agreement

A fundamentally important aspect when dealing with distributed systems is the ability of the system to form non trivial[14] agreement on a particular value or set of values. It has been shown in ([Her91]) that it is possible to implement any *wait-free* concurrent data object among a set of processes using distributed *Consensus*. Agreement also plays an important role in active replication and the *replicated state machine* approach, which was first described for synchronous systems by Leslie

[12]Sometimes also called process controlled crash

[13]In the partitionable group membership model [CKV01] show that *precise membership* is as strong as a $\diamond\mathcal{P}$ failure detector.

[14]For instance if all processes always return 0 when queried the basic agreement property is trivially satisfied.

Lamport ([Lam84]). Reaching agreement has been formalized through the *Consensus problem*. It is characterized by three properties that every algorithm solving Consensus has to satisfy and that are typically formulated as follows: ([Fuzz08])

1. *Validity*: If a process decides a value v, then v was proposed by some processes
2. *Agreement*: No two correct processes decide differently
3. *Termination*: Every correct process (eventually) decides some value

The properties *Validity* and *Agreement* are usually referred to as s*afety properties* because they ensure that the algorithm can't perform unwanted actions. *Termination* is often also called *Liveness* as this property guarantees that the system will eventually make progress and is not forever stuck in an undecided state ([Fuzz08]). Fisher, Lynch and Patterson have shown that in an asynchronous system it is impossible to deterministically reach agreement, even if only a single process can fail and communication is reliable ([FLP85]). This result stems from the fact that in an asynchronous system it becomes impossible for a process to discern if another process has crashed or is merely very slow. Ensuring both safety properties under these circumstances means that liveness can no longer be guaranteed, as processes might have to wait indefinitely for an answer from a crashed process. To alleviate this situation and be able to reach agreement some form of synchrony needs to be introduced into the system.

([DDS87,DLS88]) have defined various models of partial synchrony in which Consensus is solvable. In ([CT96]) Chandra and Toueg introduce the notion of *unreliable failure detectors* and show how they can be used to solve Consensus in asynchronous systems. This still requires the failure detectors themselves to work in a model of partial synchrony, however it provides an abstraction to the otherwise asynchronous algorithm so that no other timing assumptions need to be made. As already mentioned in the previous chapter on failure detectors, Chandra and Toueg show that the weakest class of failure detectors required to solve Consensus is that of $\diamond\mathcal{W}$ [15] if a majority of processes does not fail. Hence the question whether Consensus can be reached is reduced to the question if a $\diamond\mathcal{W}$ failure detector can be implemented in the given system model.

It is important to point out that solving Consensus in either $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ requires that at least $\lceil(n + 1)/2\rceil$ processes are correct or, expressed otherwise, less than half the processes $f < \lceil n/2\rceil$ are allowed to fail. With failure detectors in $\mathcal{P}$ and $\mathcal{S}$ there is at least one correct process that is never suspected and hence can fulfil the role of the coordinator for reaching agreement. In $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ however the failure detector is allowed to suspect any processes for some time until it eventually stops suspecting at least one correct process. During this initial time of indecision the failure detectors could (logically) partition the processes into two or more groups, where one group suspects all processes of the other and vice versa. If Consensus is allowed to be formed during such a time of partitioning, partitions may reach agreement on different values thereby invalidating the *agreement* property of Consensus. Requiring a majority of processes to be correct prevents other (smaller) partitions to agree on a different value. Obviously such a requirement implies that the actual failure of a majority of processes will make it impossible to reach agreement.

[15] In fact the proof they present is for $\diamond\mathcal{S}$, however they also give a reduction algorithm that can transform $\diamond\mathcal{W}$ into $\diamond\mathcal{S}$

There exist a number of agreement problems such as *total order (atomic) broadcast*, *atomic commitment* or *group (set) membership* which are all related to the abstract Consensus problem and are hence subject to the FLP *impossibility result* in an asynchronous system model ([GS01]). In ([GS01]) Guerraoui and Schiper describe a "Generic Consensus Service" where, by reducing said agreement problems to a variant of the Consensus problem, a single generic Consensus layer is sufficient to derive such agreement problems. Many recent works on group communication systems take into consideration that the group membership problem is expressible as a variant of the Consensus problem and hence use Consensus as a basic building block on top of which group membership is implemented. ([GHRT01]) for instance employ a *general agreement framework* to build the membership service and other components of a primary component group membership service.

2.2.5 Multicast Protocols for Group Communication

Protocols and Stacks

A protocol can be described as "a distributed algorithm that solves a specific problem in a distributed system, e.g. a TCP protocol solves the reliable channel problem. A protocol is implemented as a set of identical *protocol modules* located on different machines." ([RWS06]). Usually a distributed service will rely on different protocols to provide its guarantees. These individual protocol modules may interact with each other and a set of such modules is called a *protocol stack*. "A *stack* is a set of protocol modules (of different protocols) that are located on the same machine. Note that, despite its name, a stack is not strictly layered, i.e. a protocol module can interact with all other protocol modules in the same stack, not only with the protocol modules directly above and below." ([RWS06]).

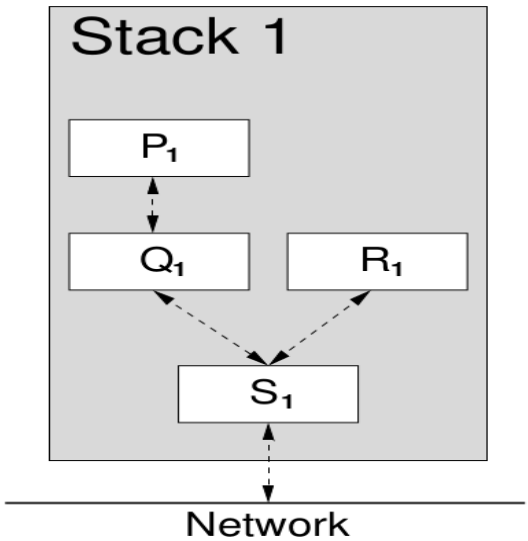

Fig. 3: Example of a protocol stack [RWS06]

Sending a message to multiple recipients

Within a distributed system it is desirable to have primitives where message sending is not just targeted to a single process, but rather to the entire set of processes or a subset thereof. *Multicast* and *broadcast protocols* present such primitives and also provide different message delivery and ordering guarantees. A *multicast* is the sending of a message to the members of some particular *group* while a *broadcast* is targeted at all the processes in the system. Effectively a broadcast is a special case of a multicast where the targeted group contains all processes of the system. A *group* is a subset of the processes of the system and a distributed system can contain multiple overlapping groups. ([HT94]) Groups can be *open* or *closed*, the former allowing non-members of the group to send messages to it while the latter requires senders to also be part of the group. Within *static groups*[16] the set of processes that constitute the group does not change, while *dynamic groups* allow processes to be added or removed. *Dynamic groups* present a problem called the *group membership problem* where agreement needs to be formed on the set of processes that make up the group every time processes are added or removed. Each (agreed) upon set of processes is called a *view*, as it represents the view of the processes that make up the group at that particular point. Many specifications on multicast[17] protocols often factor out the group membership problem by assuming *static groups.* Special care may need to be taken when the membership of a dynamic group changes in order to ensure that the guarantees provided by multicast protocols are upheld. One such approach is *virtual synchrony*, where processes agree upon the set of messages they have seen in a particular *view* before moving to the next. (Dynamic) group membership and virtual synchrony is discussed in the next section (2.2.6) in more detail.

Multicast protocols are generally not just implemented as an abstraction to conveniently be able to send a message to an entire group, but rather to addresses an important issue encountered in distributed systems where processes may fail. During the multicasting of a message the sending process may crash so that some processes will have received the message while others won't[18]. Such inconsistencies are undesirable and complicate the development of fault-tolerant distributed software ([HT94]). Beyond attempting to ensure the *reliable* delivery of multicast messages numerous fault-tolerant multicast and broadcast protocols have been developed over the past decades for a variety of generalized or specific use cases and with different guarantees in regards to message delivery and ordering.

In ([HT94]) Hadzilacos and Toueg present modular specifications for various types of fault-tolerant broadcasts and multicasts in a *static group* model. The weakest considered primitive is *reliable broadcast* (respectively *reliable multicast*) from which they define progressively stronger guarantees. Some of the multicast protocols for asynchronous systems presented in their work are hereby covered:

[16]A static group implies that the members always are in agreement on the set of processes that make up the group, however the status (faulty or not) of each process still needs to be determined.

[17]From here on the term multicast protocol is used to describe both multicast and broadcast protocols unless otherwise specified

[18]As we have covered in the Fundamental Concepts it is impractical to provide reliable channels so a message may be lost if the sending process fails. Similarly it is impractical to provide multicast primitives as an atomic action where a message is received either by all processes or none.

Reliable Multicast

Reliable multicast is the most basic fault-tolerant multicast and is defined in terms of both *multicast* and *deliver* primitives. It guarantees the three properties that 1) all *correct* processes of the group agree on the set of messages they deliver, 2) this set of messages contains all messages multicast by *correct* processes and 3) no spurious messages are ever delivered. More formally ([HT94]) describe reliable multicast in terms of a validity, agreement and integrity property:

- *Validity:* If a correct process multicasts a message *m*, then some correct process in *group(m)* eventually delivers *m* or no process in that group is correct.
- *Agreement:* If a correct process delivers a message *m*, then all correct processes in *group(m)* eventually deliver *m.*
- *Integrity:* For any message *m*, every correct process *p* delivers *m* at most once, and only if *p* is in *group(m)* and *m* was previously multicast by *sender(m).*

Reliable multicast does not give any guarantees on the *order* in which these messages are delivered. It is generally desirable to receive all messages from a correct single sender in the order in which they were sent. Such an ordering can, for instance, be achieved by assigning to each message a sequence number so that the combination of *sender(m)* and *m*'s sequence number forms a unique ordered identifier. Under the assumption that channels are quasi-reliable a correct process will eventually receive all messages sent by a correct sender. The receiver merely has to ensure that they do not deliver these messages prematurely in the wrong order by buffering them until this so called *FIFO guarantee* can be met.

FIFO Multicast

FIFO (first-in-first-out) multicast is a reliable multicast with the following additional property: ([HT94])

- *Global FIFO Order:* If a process multicasts a message *m* before it multicasts a message *m',* then no correct process in *group(m)* delivers *m'* unless it has previously delivered *m.*

As opposed to FIFO broadcasts where the group is always the same, FIFO multicast may take into consideration that messages can be addressed to different groups, allowing for a weaker ordering guarantee that only requires FIFO ordering for messages sent to the same group:

- *Local FIFO Order*: If a process multicasts a message *m* before it multicasts a message *m*' such that *group(m') = group(m)*, then no correct process delivers *m*' unless it has previously delivered *m.*

FIFO multicast can only preserve the context of messages from a single sender and does not take multicasts from other processes into consideration. As an example consider a scenario where processes respond to requests that are multicast by a server by multicasting their local time. FIFO ordering will ensure that no process observes a response from a single process with a time that is less than any previous time that single process has multicast. However the *causal order* in which responses in regards to requests were sent is not preserved. This means that a process may deliver responses to a query before having seen (delivered) the corresponding request from the server. *Causal multicast* preserves this relationship and in regards to this example processes would only be able to deliver the responses to a query once they have delivered the corresponding request. The request *causally precedes* the response.

Causal Multicast

Causal multicast is a reliable multicast that furthermore satisfies: ([HT94])

- *Global Causal Order*: If the multicast of a message *m* causally precedes the multicast of a message *m'*, then no correct process in *group(m)* delivers *m*' unless it has previously delivered *m*.

where causal precedence is defined as:

Causal Precedence : Step *e causally precedes* step *f*, denoted $e \rightarrow f$, if and only if:

1. the same process executes both *e* and *f*, in that order, or
2. *e* is the broadcast of some message *m* and *f* is the delivery of *m*, or
3. there is a step *h*, such that $e \rightarrow h$ and $h \rightarrow f$.

As with FIFO multicast causal ordering can also be considered globally or only for messages that are part of the same group. For the latter the following weaker definition of *local causal order* ([HT94])

- *Local Causal Order*: If the multicast of a message *m* causally precedes in *group(m)* the multicast of a message *m*', then no correct process delivers *m*' unless it has previously delivered *m*.

Causal multicast ensures that messages that are related through causal precedence are delivered in their proper order. However for some applications it can be required that all correct processes deliver the same sequence of messages in the same order. Ensuring such total order is especially useful for multicasting commands to a group of replicas to keep their internal state consistent.

A multicast protocol that guarantees correct processes deliver all messages in the same order is called *total order* or *atomic* multicast.

Atomic Multicast

Hadzilacos and Toueg describe three different types of atomic multicast with increasingly stronger guarantees. The following properties each extend those of reliable broadcast: ([HT94])

- *Local Total Order*: If correct process *p* and *q* both deliver messages *m* and *m*' and *group(m) = group(m')*, then *p* delivers *m* before *m*' if and only if *q* delivers *m* before *m*'.

With this guarantee total order only needs to be upheld for messages that are multicast to the same group. Processes belonging to multiple groups may observe the same set of messages from both group *G* and *G'* in different interleaving orders as long as each individual message from each group is in total order in respect to the other messages from that group.

- *Pairwise Total Order*: If correct processes *p* and *q* both deliver messages *m* and *m*', then *p* delivers *m* before *m*' if and only if *q* delivers *m* before *m*'.

Pairwise Atomic Multicast ensures that processes which share multiple groups also have to deliver messages from these different groups in total order in respect to each other. This however does not completely cover all potential undesirable situations. In scenarios with three or more overlapping groups cycles in message delivery order can occur. The following example is given in ([HT94]) :

> "For example, consider three groups, $G_1 = \{p,q\}$, $G_2 = \{q,r\}$ and $G_3 = \{r,p\}$. Note that the intersection of any two of these groups consists of exactly one process. The messages m_1, m_2 and m_3 are multicast to groups G_1, G_2 and G_3, respectively. Pairwise Total Order allows process p to deliver m_3 before m_1, q to deliver m_1 before m_2, and r to deliver m_2 before m_3."

To alleviate these potential cycles *global atomic multicast* is a reliable multicast with the following property:

- *Global Total Order*: The relation < is acyclic, where the relation < is defined on the set of messages delivered by correct processes as: $m < m'$ if and only if any correct process delivers m and m', in that order.

So far the definitions for atomic multicast have only taken into account that messages have to be delivered in the same sequence at all correct processes. The three variants of atomic multicast can be extended to additionally account for either FIFO or causal ordering of messages in both local and global variants, resulting in both six different types of FIFO and causal atomic multicast.

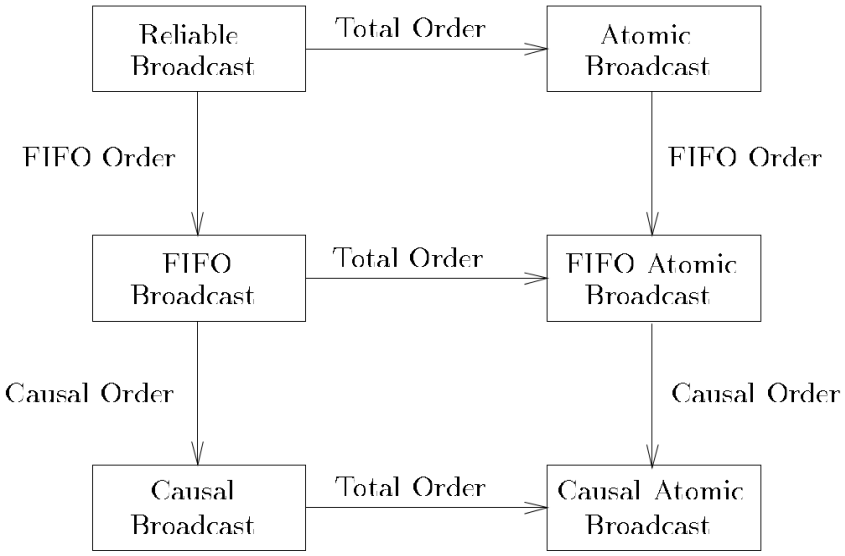

Fig. 4: Relationship among Broadcast Primitives [HT94]

Uniformity

The presented guarantees of reliable multicast (and stronger guarantees based on it) only apply to correct processes, placing no restrictions on the behaviour of those that are faulty. They can be strengthened with the following properties to also account for faulty processes:

- *Uniform Agreement*: If a process (whether correct or faulty) delivers a message m, then all correct processes in *group(m)* eventually deliver m.
- *Uniform Integrity*: For any message m, every process (whether correct or faulty) p delivers m at most once, and only if p is in *group(m)* and m was previously multicast by *sender(m)*.

Likewise the *ordering guarantees* (FIFO, causal and total order) can be strengthened by requiring that every process, correct or faulty, upholds these guarantees for messages they delivered. Providing *uniformity* generally presents a harder problem than using the *non-uniform* counterpart.

Message Stability

Different message ordering and delivery guarantees have different "costs" associated with them. For instance a FIFO multicast protocol has to hold back received messages if it is missing previous messages from the sequence, slowing down transmission speeds. Strong guarantees such as those provided by atomic multicast can incur several rounds of (lower level) message communication between members of the group for each multicast message. Guaranteeing uniformity also requires processes to hold back the delivery of messages until the property can be ensured. In this context the term *message stability* or *message safety* is of importance. A message *m* is said to be *k-stable* if *m* has been received by *k* processes ([DSU04]). In systems with an upper bound on the number of processes *f* that may crash, if a message is *f+1* stable at least one correct process will have received the message. If such a condition is met we know that the message is eventually received by all correct processes. Message stability can be used to ensure the *uniform agreement* property for multicasts. For instance a uniform reliable multicast algorithm in a system where a majority of processes of the group are always correct only has to wait for $\lceil (n + 1)/2 \rceil$ acknowledgements from other processes that they received the message to be able to deliver it. Uniformity is ensured because any process, faulty or not, can only deliver a message if it has been acknowledged by at least a majority of processes. The approach is akin to that used in the Chandra and Toueg Consensus algorithm (see [CT96]). There the requirement of a majority prevents multiple partitions to perform different actions. In this case waiting for *f+1* acknowledgements means the delivered message was seen by at least one correct process and no partition can form where all processes that have seen the message fail and no correct process learns of the (delivered) message. Incidentally the Chandra and Toueg Consensus algorithm is also *uniform*. The obvious downside to uniform reliable broadcast is that messages have to be held back until *f+1* stability is guaranteed. Furthermore the system must ensure that the number of failures at all times adheres to this upper bound or else the protocol will block when too many failures > *f* occur. Generally GCSs will indicate that a message is *safe* if it is stable for all processes of a view ([CKV01]).

Considering Dynamic Groups

As has been mentioned at the start of this section the presented specifications only cover *static groups*. Assuming a *dynamic group* model increases the complexity of multicast specifications as they have to additionally account for a changing set of members. Generally the problem of ensuring a consistent *view* of the members of a group is externalized in a specific *membership service*. Such a service provides notifications of changes in the membership in a consistent manner to all members of the group and is discussed in the next section. Without a separate membership service protocols have to individually keep track and agree upon the set processes that are currently part of the group. Membership services usually provide protocols with *views* that contain the set of processes that currently make up the group. Views are ordered in respect to each other and the succession of views describe the changes in membership over time. Considering *dynamic groups* in an asynchronous communication model means that it is possible for messages to arrive after the membership has changed. The sender of such a message might no longer be part of the current view when the multicast message is eventually received. It is hence desirable to order view changes in regards to the messages multicast within views. Furthermore specifications for fault-tolerant multicasts might apply only to the messages sent within individual views or be required to uphold across views. As an

example consider a FIFO multicast where messages are only ordered within individual views and one where FIFO ordering is also required across views. The former allows a newly *joined*[19] process to receive FIFO multicast messages of that view without any previous context while the latter requires that the process has all FIFO multicast messages from previous views.

Generally speaking, the reduced complexity of the static group model is beneficial when presenting and specifying concrete algorithmic implementations for different fault-tolerant multicasts.

2.2.6 The Group Membership Problem and Virtual Synchrony

The Problem

The previous section has briefly covered some of the aspects when talking about *groups* in distributed systems, in particular in regard to multicast protocols. A *group G* is a set of processes that form a subset $G \subseteq P$ of all the processes of the distributed system. Different groups may be overlapping and can either allow the addition and removal of processes (*dynamic group*) or have a fixed set of members (*static group*). In addition to the issue of determining whether or not processes have crashed, dynamic groups present a problem called the *group membership problem*[20].

This problem was first defined for synchronous systems in ([Cri91]) and has been investigated in the asynchronous system model for *primary-partition* ([RB91],[KT91],[MPS91],[MSMA94]) and *partitionable group membership* ([JFR93],[vRBC+93],[BDGB94],[DMS94]). Roughly speaking, the group membership problem tries to solve the issue of maintaining and agreeing on the dynamically changing set of processes that are currently correct ([ST06]). The goal is to provide a distributed *group membership service* that presents processes with a consistent *view* of the processes that constitute the group. The successive memberships of a group are called the *views* of the group ([DSU04]). Whenever this membership changes a *view change* takes place, where the group membership service notifies all members of the group of the new *view*[21].

Primary-Component Group Membership

Primary Component Group Membership or PCGM (also called primary-partition group membership) assumes that in case of partitions at most one (primary) partition may be allowed to make progress. This is to prevent the group from suffering from a so called “split brain”, where two or more partitions continue to concurrently install different views for the alleged same group. In this respect the partitioning may not just be caused by failures of the underlying communication channels but can also occur when the unreliable failure detectors of processes suspect each other and hence logically partition the network. The primary partition group membership problem resembles the Consensus problem in many aspects, as agreement needs to be formed on the set of processes that make up successive views, however it differs in at least two ways ([CHTCB95]):

[19] A joined process is one that has not been present in view *v* and is now included in the current view *v+1*

[20] Also called processor-/process-group membership problem.

[21] The membership service may also send the new view to excluded processes from the previous view in an attempt to notify them of their exclusion. This may be of value for processes that are leaving or were wrongfully suspected and need to re-join.

- In group membership, a process that is suspected to have crashed can be *removed* from the group, or even *killed*, even if this suspicion is actually incorrect (e.g. the suspected process was only very slow). The ([FLP85]) model does not speak about process removals, and it does not directly model process killing (e.g. program-controlled crashes).
- Consensus requires progress in all runs, while group membership allows runs that "do nothing" (for instance, "doing nothing" is desirable when no process wishes to join or leave the group, and no process crashes).

Partitionable Group Membership

Partitionable Group Membership allows for multiple concurrent but disjunct views of the same group to exist at any one time. Since such a model does not require agreement on a particular view by all correct processes of the group its *liveness properties* are strengthened. In particular, partitionable group membership does not have to block on view changes if a single primary partition cannot be ensured. The obvious downside of such an approach is that protocols relying on a partitionable membership service have to be able to merge different states once the partitioning ceases. As an example the replicated state machine approach, if it were implemented using partitionable group membership, would allow replicas in different partitions to diverge from another. Once the partitioning ceases all replicas need to be merged into a single consistent state. Decisions by the group that are based on previous views are problematic in partitionable group membership because it is possible that partitions decided differently. Reducing the given state machine example to a simple binary output, one partition may have decided 0 while the other chose 1. If successive steps in the computation depend on this value the merging of both partitions would require that one partition has to change previously decided values. Generally speaking the partitionable group membership model renders the specification of reliable multicast protocols and virtual synchrony more complex and difficult than primary component group membership.

Specifications

Many early specifications of primary component group membership assumed that through program-controlled crash the group membership problem was solvable in the asynchronous system model. Unfortunately these specifications proved to be incomplete ([ACBMT95,CHTCB95]) and Chandra et al. in ([CHTCB95]) showed that the impossibility result of ([FLP85]) also holds true for the primary-partition group membership problem in an asynchronous system model with crash failures, even when communication is reliable and program-controlled crash is used[22] (specifically, liveness cannot be guaranteed during all execution runs). As with other forms of distributed Consensus, employing the help of oracles or introducing other synchrony assumptions does render the primary-partition group membership problem solvable.

In ([CKV01]) Chockler et al. present an extensive survey of works on group communication systems (GCSs) and provide a set of specifications that can be combined to present the guarantees of most existing GCSs. Group communication systems offer both membership and reliable multicast

[22][CHTCB95] specify a Weak Group Membership problem (WGM) which any realistic group membership service should provide and show that impossibility results both with and without program-controlled crash hold.

services. It should be pointed out that in ([CKV01]) it is not the goal of the authors to provide a single unifying specification. Rather they present modular “components” that can be combined to classify and describe different GCSs or build concrete specifications. Real-world group communication systems often follow best effort approaches where situations such as the previously described inability to guarantee liveness for all execution runs is an acceptable drawback, as the occurrence is rare and often avoidable. The following properties from their work cover some of the more important aspects of GCSs, but are by no means a complete iteration. Only the informal descriptions are used and the reader is pointed to ([CKV01]) for formal definitions and a complete specification of their classification framework.

Safety Properties

Some of the basic safety properties generally found in membership services are defined by ([CKV01]) as follows:

- Property 3.1 (Self Inclusion). If process p installs view V, then p is a member of V
- Property 3.2 *(Local Monotonicity).* If a process *p* installs view *V* after installing view *V'* then the identifier of *V* is greater than that of *V'*.
- Property 3.3 *(Initial View Event).* Every **send**, **recv**, and **safe_prefix** event occurs within some view.

  Where **send***(p,m)* means process *p* sends message *m*, **recv***(p,m)* means process *p* receives message *m* and **safe_prefix***(p,m)* indicates to *p* that message *m* is *stable;* that is, all members of the current view have received this message from the network.

In a primary component group membership service the following additional property is introduced:

- Property 3.4 *(Primary Component Membership).* There is a one-to-one function *f* from the set of views installed in the trace to the natural numbers, such that *f* satisfies the following property. For every view *V* with $f(V) > 1$ there exists a view *V'*, such that $f(V) = f(V') + 1$, and a member *p* of *V* installs *V* in *V'* (i.e., *V* is the successor of *V'* at process *p*).

Liveness Properties

Liveness properties can be difficult to specify for a GCS:

“Liveness is an important complement to safety, since without requiring liveness, safety properties can be satisfied by trivial implementations that do nothing. However, it is challenging to specify GCS liveness properties that are sufficiently weak to be implementable and yet are strong enough to be useful.” ([CKV01])

In this context Chockler et al. only present liveness properties for partitionable GCSs, stating for PCGM that:

“... the liveness of such a service is dependent on the specific implementation and the policy it employs to guarantee Property 3.4 (Primary Component Membership).” ([CKV01])

Their liveness property for *partitionable group membership* is as follows:

- Property 10.1 (Liveness). If the failure detector behaves as $\diamond\mathcal{P}$, then for every stable component S, there exists a view V with the members set S such that the following four properties hold for every process p in S.
  1. *Membership Precision*: p installs view V as its last view.
  2. *Multicast Liveness*: Every message p sends in V is received by every process in S.
  3. *Self Delivery:* p delivers every message it sent in any view unless it crashed after sending it.
  4. *Safe Indication Liveness:* Every message p sends in V is indicated as safe by every process in S.

Where a *stable component* is defined as

"... a set of processes that are eventually alive and connected to each other and for which all the channels to them from all other processes (that are not in the stable component) are down." ([CKV01])

Obviously in PCGM there can be at most one such stable component as there is at most one primary-component. The above liveness specifications can be adapted for PCGM if the further restrictions of ensuring a primary partition are also factored in, but obviously depend on the specific implementation. Here the opportunity is taken to point out why such liveness properties can be difficult to specify for primary component group membership. *Agreement* needs to be reached on the successive views of the PCGM and it is the employed agreement mechanism that directly influences liveness.

An implementation of PCGM may use an *almost perfect failure detector* in combination with an agreement protocol such as that described by Ricciardi and Birman ([RB93]) which is similar to three phase commit (3PC)[23]. Here solving the more difficult problem of non-blocking atomic commit is addressed by seemingly[24] relying on a failure detector in $\mathcal{P}$. In such an approach liveness greatly depends on how accurate failure suspicions are as incorrect suspicions effectively force correct processes to crash. On the other hand, using a *Consensus* algorithm such as described by ([CT96]) for agreeing on views can ensure liveness as long as a majority of processes partaking in the Consensus is correct (assuming the network eventually has a long enough period of stability to permit Consensus being reached). Naively, even a single process could issue views for PCGM so the liveness of the membership service would depend on the liveness of that single process. GCSs may use their own, specific form of *agreement* for PCGM, possibly without formal specifications on liveness, or rely on well known agreement protocols. It is clear that these issues greatly complicate a generalization on the liveness of primary component group membership services and specifications are closely tied to the particular implementation or chosen approach.

[23]This is the approach presented by Birman et al. in ISIS [RB93].
[24]Liveness cannot be guaranteed in asynchronous systems and still falls under the impossibility result of [FLP85]

Virtual Synchrony

The concept of *virtual synchrony* was first introduced by Birman and Joseph for primary component group membership.

“In an environment of virtual synchrony, the sequence of events (broadcasts and process failures) observed by each process is equivalent to one that could have occurred if the processes were operating in an environment where broadcasts and failure detection were synchronous.” ([BJ87])

Chockler et al. define the virtual synchrony property as ([CKV01])[25]:

- Property 4.5 *(Virtual Synchrony)*. If processes *p* and *q* install the same new view *V* in the same previous view *V'*, then any message received by *p* in *V'* is also received by *q* in *V'*.

What this property means is that if any two processes both directly progress from the same common view to the one installed right afterwards they will agree on the set of messages received in the previous view. Virtual synchrony can simplify the implementation of reliable multicast protocols and offer performance improvements such as allowing relaxed message ordering ([BJ87,Bir10]).

Virtual synchrony also implies another property called *same view delivery*, which requires that messages are delivered within the same view at every process. ([CKV01]) define the same view property as:

- Property 4.4 *(Same View Delivery)*. If processes *p* and *q* both receive message *m*, they receive *m* in the same view

It may be desirable to further restrict the same view delivery guarantees so that messages that were sent in a view also have to be delivered in that same view. This property is called sending view delivery and is specified by ([CKV01]) as follows:

- Property 4.3 *(Sending View Delivery).* If a process *p* receives message *m* in view *V*, and some process *q* (possibly $p = q$) sends *m* in view *V*, then $V = V$

In ([FR95]) Friedman and Renesse show that to be able to implement sending view delivery, what they refer to as *strong virtual synchrony*, without violating other properties such as *virtual synchrony* or *self delivery*, processes have to eventually stop sending new messages within the old view and block. Blocking is required so that agreement on the set of messages sent within the view can eventually be reached. Another property in the context of virtual synchrony is the *transitional set.* It is the set of processes for which the concept of virtual synchrony upholds after a view change. ([CKV01]) define it as:

- Property 4.6 *(Transitional Set).*
    1. If process *p* installs a view *V* in (previous) view *V'*, then the transitional set for view *V* at process *p* is a subset of the intersection between the member sets of *V* and *V'*.
    2. If two processes *p* and *q* install the same view, then *q* is included in *p*'s transitional set for this view if and only if *p*'s previous view was also identical to *q*'s previous view.

[25] In this context *receive* means the message was *delivered* to the application/process

The transitional set is more important for *partitionable group membership* where processes could have been part of different intermediate views and hence cannot simply deduce the processes for which virtual synchrony upholds simply by forming the intersect of processes in V and V'.

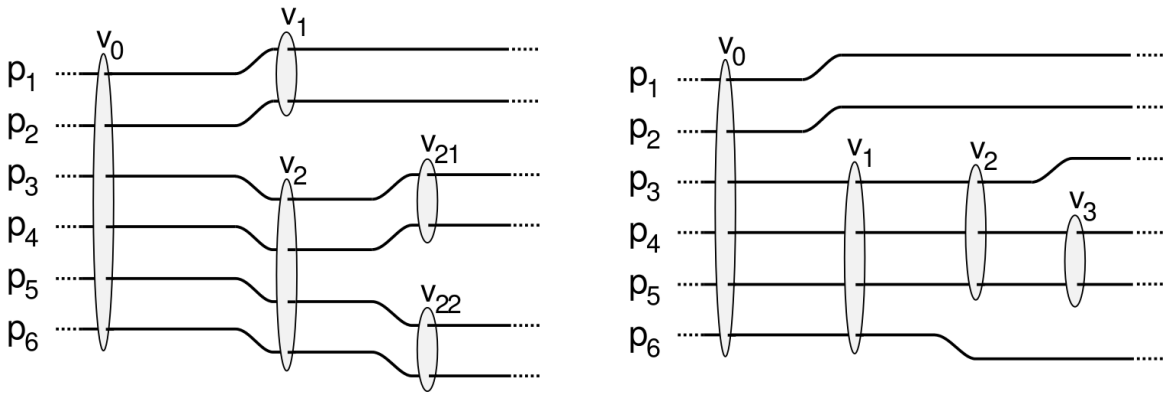

Fig. 5: Concurrent Views (left) and Unique View Sequence (right). [Mal96]

Virtual synchrony can be seen as the dynamic group counterpart of reliable broadcast in static systems ([DSU04]). It addresses some of the issues of how processes order messages in regards to the views that are delivered through the group membership service. GCSs usually provide additional multicast services with guarantees that extend those of virtual synchrony: “These service types involve two kinds of guarantees: ordering and reliability. The ordering properties restrict the order in which messages are delivered, and the reliability properties extend the corresponding ordering properties by prohibiting gaps or “holes” in the corresponding order within views” ([CKV01]). Some of these guarantees have already been described in the section on multicast protocols (2.2.5 Multicast Protocols for Group Communication). Ordering guarantees can also apply across different multicast protocols and groups. Messages of a FIFO reliable multicast might still have to be delivered in total order in regards to the messages of an atomic multicast to the same group. Processes belonging to an intersection of different groups may also need to order multicasts to individual groups with respect to each other. Ordering and reliability constraints greatly depend upon the intended usage scenario. Specifications need to be chosen wisely as stronger guarantees usually incur greater costs and have a negative impact on the performance of the resulting GCS. The reader is referred to ([CKV01]) and ([HT94]) for a more detailed discussion on the topic.

Table VI. Summary of Safety Properties of the Membership and Multicast Services

Basic Properties	Optional Properties
Self Inclusion	Primary Component Membership
Local Monotonicity	Sending View Delivery
Initial View Event	Virtual Synchrony
Delivery Integrity	Transitional Set
No Duplication	Agreement on Successors
Same View Delivery	

Table VII. Properties of Different Ordered Multicast Services and of Safe Message Indications

FIFO Multicast	Causal Multicast
FIFO Delivery	Causal Delivery
Reliable FIFO	Reliable Causal
Totally Ordered Multicast	Safe Indications
Strong Total Order	Safe Indication Prefix
Weak Total Order	Safe Indication Reliable Prefix
Reliable Total Order	

Table VIII. Summary of Liveness Properties

Basic Properties	Optional Properties
Membership Precision	Safe Indication Liveness
Multicast Liveness	Termination of Delivery
Self Delivery	Membership Accuracy

Fig. 6: Basic and optional properties for GCSs as presented in [CKV01]

PCGM can act like an almost perfect failure detector

Membership services are faced with a difficult problem in the non-partitionable dynamic group model. It is in the nature of unreliable failure detectors that correct processes can be wrongfully suspected. Relying on such failure detectors may therefore inadvertently exclude non-faulty processes from the group, even if they are un-suspected at a later time. In the partitionable group membership model component groups can evolve concurrently, so the exclusion of processes from a view does not necessarily imply their failure, but may mean that they have partitioned and might later rejoin. PCGM on the other hand requires that at most one primary component exists. Any processes that is not part of this primary view cannot be part of a valid view of the group. Excluded processes are hence forced to block (or even crash) and need to rejoin the group to again be valid members.

Loosely speaking PCGM reduces the complexity of protocols, which do not have to consider concurrent groups and merging of their partitionable counterparts at the cost of forcing correct but wrongfully suspected processes to block and lose virtual synchrony. Furthermore in PCGM the entire group can be prevented from making progress if no single primary partition can be ensured. PCGM acts similarly to an *almost perfect failure detector* $\mathcal{AP}$, where incorrect suspicions lead to actual failures and reduce the resilience of the system. In this case however it will depend on the concrete specification of the membership service how severe a wrongful exclusion may be[26].

[26]It is clear that an incorrect suspicion and subsequent removal of correct processes can lead to situations where the newly formed view *v+1* is no longer resilient enough and the failure of processes leads to a situation where no agreement on the

PCGM and Consensus

In real systems some assumptions such as *reliable* or *quasi-reliable* channels are not realistic and actually introduce problems that require a failure detector in $\mathcal{P}$ to be solvable:

“This issue can be formalised by the time-bounded buffering problem [8]. Let m be a message in the output buffer of process p that must be sent to process q: time-bounded buffering ensures that p eventually deletes m from its output buffer. The problem cannot be solved in an asynchronous system model, neither in an asynchronous system model augmented with failure detectors of class $\mathcal{S}$ or class $\diamond\mathcal{P}$ [5,8]. The same holds for Reliable Broadcast over fair-lossy channels. Real system overcome this impossibility by relying on program-controlled crash” ([Sch02])

While PCGM, when used as a failure detector, behaves similar to using program-controlled crash it itself is an agreement problem and can be solved without having to rely on such measures. In ([GS01]) Guerraoui and Schiper present a generic agreement framework and show that the group membership problem can be reduced to Consensus, which is solvable with a $\diamond\mathcal{W}$ failure detector. *Phoenix* ([Mal96]) employs a modified version of the Consensus algorithm in ([CT96]) for agreeing on group membership. Greve et al. ([GHRT01],[GlN04]) also present an approach where a primary component asynchronous group membership service is constructed as an instance of a generic agreement framework. In ([ST06]) Schiper and Toueg express group membership as a special case of the *set membership problem.* They argue that the group membership problem actually tries to solve two orthogonal problems, namely 1) determining the set of processes that are correct and 2) ensuring that processes agree on the successive values of this set. It is proposed that failure detection for the membership and failure suspicions for solving agreement on the membership are decoupled. When speaking on the trade-offs between group membership and failure detection Schiper presents the following argument ([Sch02]):

“Developing complex membership protocols — instead of reducing membership to Consensus — had an indirect consequence. It has hidden the benefit of decoupling “failure detection” from “membership exclusion”. While a group membership service gives a consistent information about the state of processes (correct or not), failure detection provides an inconsistent information. It may sometimes be sufficient and less costly to rely on inconsistent failure detection information rather than on consistent group membership information.”

Using Consensus with unreliable failure detectors to agree upon the membership rather than relying on the stronger failure detector $\mathcal{AP}$ and a specific membership protocol has several advantages. The liveness of the system is strengthened during periods where agreement needs to be formed on the next view. Process-controlled crash is more likely to cause a permanent blocking of the group when forced failures make it impossible to guarantee a single primary component. Incorrect suspicions of the failure detector for Consensus merely delay the protocol from reaching a decision unless a majority of processes actually fail. The Consensus protocol can also weigh votes for the exclusion of a suspected process so an incorrect suspicion may not necessarily remove that process. For instance a quorum of the votes cast during a Consensus instance could be required to suspect a process for it to be actually removed from the view. Contrasting this a single wrongful suspicion with an $\mathcal{AP}$ failure detector leads to the subsequent failure and removal of the suspected process. In this context

next view *v+2* can be reached, effectively blocking the entire group forever while the correct but excluded members are waiting to rejoin.

([GlN04]) discern between the strategies of *progression by safe group*, where the computation within the group requires the participation of all correct processes and *progression by long-lived* group, where the evolution of computation requires only the participation of a subset of the correct processes.

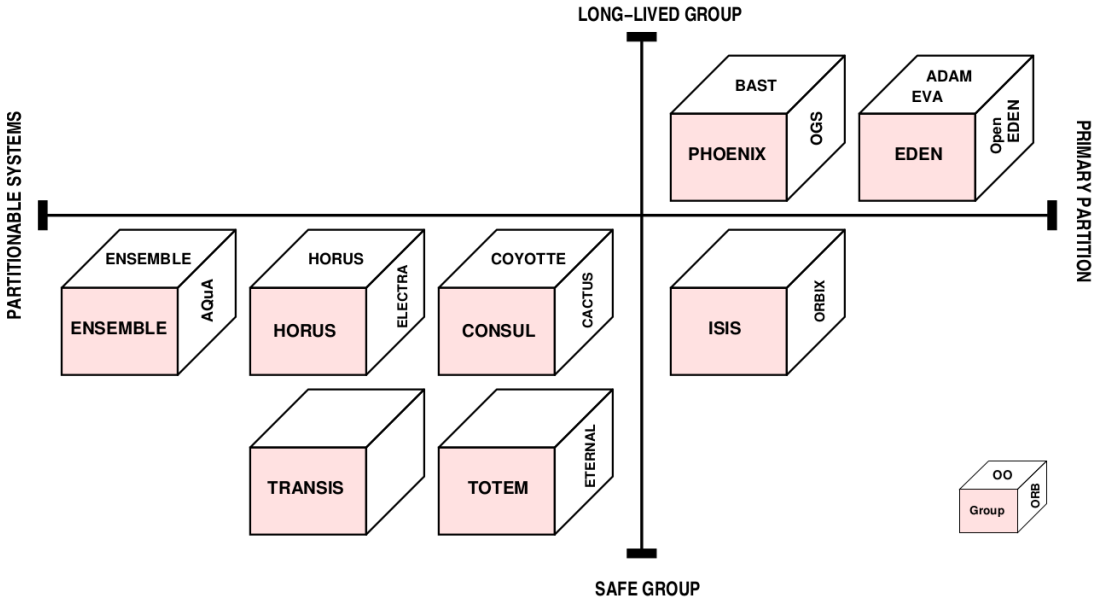

Fig. 7: Group systems by type of evolutionary approach and group membership partition [GlN04]

Consensus is a basic building block of GCSs especially for PCGM and basing the membership service on such a component can greatly reduce its complexity and increase modularity. Generic or modular Consensus services that can provide instances for specific agreement problems have been described for instance by ([GS01],[HMRT99],[SRSD08]). An algorithm solving (primary component) group membership based on Consensus can be expressed easily in a few lines of code which makes it less error prone and easier to understand. The following pseudo code example (Fig. 8) is presented in ([Sch02]).

Algorithm 1
Solving group membership among *current-view* by reduction to Consensus (code of process *p*)

1: $v_p \leftarrow$ *current-view \ suspected-processes ;*
2: $decision_p \leftarrow Consensus(v_p)$ *;*
3: {execute Consensus among *current-view;* v_p is the initial value for Consensus}
4: *new-view* $\leftarrow decision_p$ *;*

Fig. 8: Solving group membership through Consensus

Choosing Partitionable or Primary-Partition Group Membership

Partitionable group membership is often used in systems that are deployed in wide area networks (WANs) and other scenarios where partitioning is likely to occur. As opposed to PCGM the partitionable model can allow all partitions to make progress, however this increased liveness has a cost, namely the different partitions can have divergent states that need to be merged once the partitioning subsides. Developers building applications on top of partitionable group membership need to be fully aware of this discrepancy between partitions and have to design their systems accordingly. Primary component group membership provides stronger consistency at the cost of liveness. In systems using PCGM only a single notion of the progression of views exists for valid members of the group. This *agreement* on the sequence of views and their content can simplify protocol and application development. There have been many different specifications for both PCGM and partitionable group membership that try to mitigate some of the issues inherent to each model (refer to [CKV01] for an overview of works and specifications).

2.3 Group Communication Systems and Protocol Composition Frameworks

2.3.1 Group Communication Systems (GCSs)

So far only individual components or aspects of *group communication* such as group membership or different types of multicast have been presented. Group communication should be understood as a combination of different components and services that interact to provide various reliable communication primitives for groups of processes.

“Group communication is a programming abstraction that allows a distributed group of processes to provide a reliable service in spite of possible failures within the group. Group communication encompasses broadcast protocols (e.g., reliable broadcast, atomic broadcast), membership protocols, and agreement protocols. Group communication is a middleware technology, lying between an application layer and a transport layer. Developing and maintaining a group communication middleware is a non-trivial, error-prone and complex task.” ([BFM+06])

The term *Group Communication System* (GCS) refers to implementations or concrete designs that provide group communication primitives. Mena ([Men06]) differentiates between monolithic and modular GCSs. Monolithic systems do not allow for easy customization and adaptation and components are tightly coupled. Modular systems on the other hand have a more flexible composition of different, off-the-shelf components that can easily be adapted or extended. While monolithic systems do not require a protocol composition framework modular GCSs either employ an ad-hoc or general-purpose composition framework ([Men06]). Early group communication systems such as *ISIS* ([BJ87,RB91]), *Phoenix* ([Mal96]), *RMP* ([WMK95]) and *Transis* ([DM96]) present themselves as more or less monolithic constructs. They have a tight coupling of the different components, preventing easy modification and reusability ([Men06]). Since then the design of group communication middleware has increasingly become more modular to allow for greater flexibility. General purpose protocol composition frameworks are often employed whose architecture does not make assumptions on the kind of protocols that are to be composed. The reader is directed towards

([CKV01]) for an extensive list of works covering both partitionable and primary-component GCSs and ([Men06]) and ([RRL07]) as well as the works presented in the next section for examples covering more modular protocol composition and group communication.

As has been discussed in section (2.2.6 The Group Membership Problem and Virtual Synchrony) the membership component of a GCS may either allow different partitions of a group to evolve concurrently (partitionable group membership) or require that only a single primary partition (primary component group membership) exists. The type of group membership will hence dictate the design of other components such as multicast protocols or how virtual synchrony is implemented and separates GCSs into two different classes, namely partitionable and primary component GCSs.

2.3.2 Protocol Composition Frameworks for GCSs

At the heart of building distributed algorithms and protocols lies the composition model that ties components together and defines how they may interact with each other to form a meaningful service. *Protocol composition frameworks* aim at simplifying the design, implementation and configuration of communication protocols ([RRL07]). Mena ([Men06]) describes them as:

"... a protocol composition framework is the infrastructure that allows a programmer to build a complex protocol out of a set of off-the-shelf building blocks: the composition. The composition can be arranged in a stack or in a graph, which is more flexible, For the external user, it is perceived as a whole (large) middleware providing the expected (complex) service. In the same way as the goal of a middleware is to make the development of distributed applications easier, the goal of a protocol composition framework is to make the development of a complex middleware easier."

Broadly speaking a *composition* is a set of *components* that interact to provide a certain functionality. A component is:

"... a higher level construct that structures definitions of the base language. A component abstraction is a parametric component that relies on another abstract component without committing to a specific implementation for it." ([BMN05])

Components can be structured hierarchically and be composed together to offer a new service implemented in the base language ([BMN05]). Composition frameworks are designed to make it easier for composers to combine different components and to form a composition. This is achieved through specifying how components are to interact with each other. Protocol composition frameworks are, as the name implies, focused on giving the composer the ability to compose *protocols*.

"A protocol provides a well defined service by exchanging data with its peers which are replicas of the protocol running on other nodes. In the setting of protocol composition frameworks, a protocol corresponds to a component (we use these terms interchangeably) and since components can be hierarchical, a protocol can also denote a protocol composition." ([BMN05])

They also aim to provide efficient execution environments for protocol compositions and promote the design of communication services in a modular way ([RRL07]). Protocol composition frameworks implement runtime services that support the exchange of data and control information among components. Individual protocol modules offer certain guarantees and communication

mechanisms and a protocol module may itself rely on other protocol modules to provide the specified service. Such protocol compositions are either composed as a stack, where there is exactly one protocol within a given layer, or as a graph[27] where different protocol modules may interact more freely to provide a given service. Protocol compositions are executed on different processes and interact with each other through message passing[28]. Specialized protocol modules in the composition usually offer this messaging functionality. Generally the protocol modules within a composition interact with each other asynchronously through events. This focus on asynchrony can be attributed to the (usually asynchronous) message passing interaction model between protocol compositions and may allow for a greater decoupling of protocol modules ([Men06]). Protocol compositions are primarily *symmetric*, where each process runs an identical instance of a given composition. In this context the *peer interaction* pattern is often encountered:

"A particular recurrent interaction pattern in symmetric compositions is called peer interactions: a protocol module communicates with its remote peers using the service offered by a lower-level protocol module. Peer interactions have always been the core idea behind protocol stacks (a universally known example is the ISO/OSI architecture [ISO96]). The purpose is to build protocols as incremental layers of abstraction: a protocol module, together with all its peer modules, implements a distributed service that is in turn used by the upper neighbor module (and its peers) to provide a higher-level service, and so forth." ([Men06])

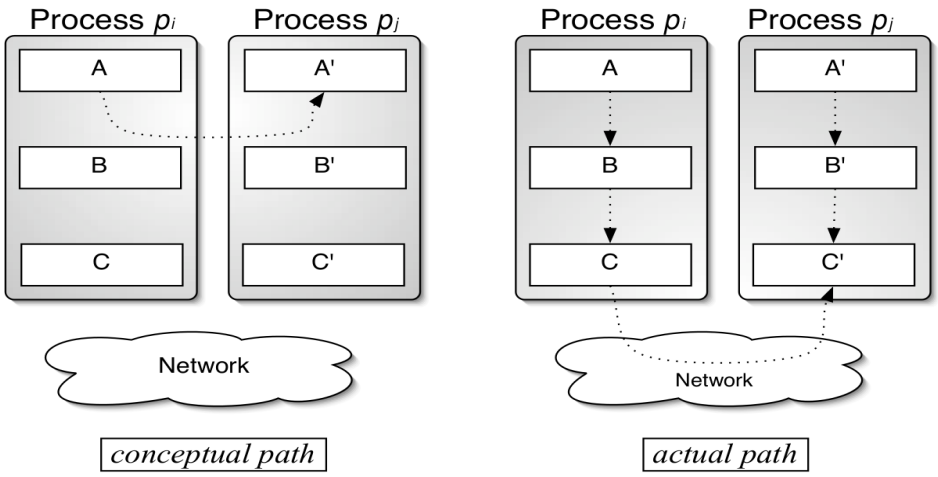

Fig. 9: Example of peer interaction from *A* to its peer *A'* [Men06]

Recent developments on protocol composition frameworks have moved towards *micro-protocol frameworks* where a protocol is itself composed of multiple micro-protocols. The interaction schemes employed vary, but are generally event based and their composition is shifting towards graph structures for protocol layering. State-of-the-art research places its focus on easy and on-the-fly reconfiguration of protocol compositions and more concise ways of modelling protocol layer interactions. More attention is also put towards the concurrency models used in protocol composition frameworks as multiprocessor architectures are becoming the de-facto standard as well as strengthening formal approaches and correctness verification ([BMF+6]).

[27]Graph-based protocol compositions are nevertheless also often referred to as (protocol) stacks.
[28]Other interaction schemes such as *distributed shared memory* are also possible but not considered in this work.

Some of the following presented protocol composition frameworks are often cited as being amongst the most relevant while others are specifically interesting in the context of this thesis, namely to be used in an educational scenario. More complete listings of relevant works for modular GCSs and protocol composition frameworks can be found in ([Men06],[RRL07]).

Cactus and the *x*-kernel

Cactus ([WHS01]) is an extension to the *x*-kernel protocol framework ([HP91]) that allows for a single *x*-kernel protocol called *composite protocol* to contain different micro-protocols, which can communicate with each other in the same protocol using events. Microprotocols in Cactus must always exist in the context of a composite protocol. Events stay contained within that composite protocol and cactus allows for dynamic binding of event handler to events which permits dynamic reconfiguration of services.

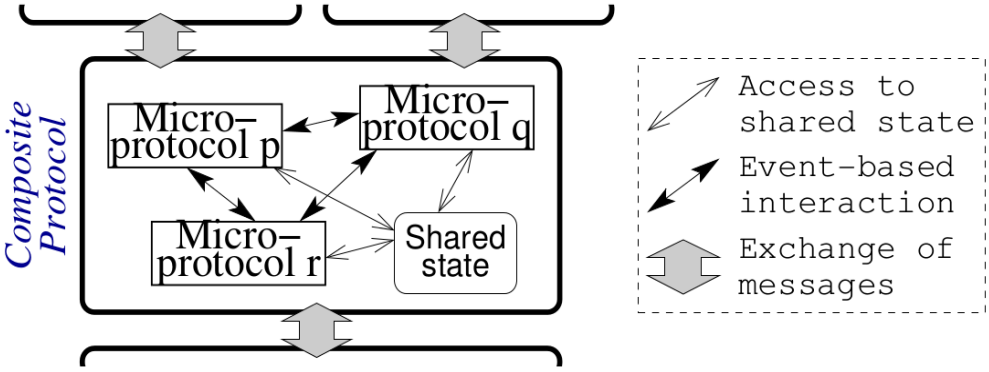

Fig. 10: Example of Cactus composite protocol [Men06]

The specification of the *x*-kernel does not require that protocols are composed strictly as a stack and allows for hierarchical graph-based compositions. To be able to decide to which upper layer protocol a message must be routed, protocols have to provide a *demux* function. Such functionality can for instance be provided through a protocol header that is pushed onto messages as they move down through the composition and can then be used in the *demux* function on the receiver's side. This methodology has many similarities to the header-driven approach proposed by Bünzli et al. ([BMN05],[BFM+06]), which will be discussed later in this work. From a concurrency perspective *x*-kernel and cactus use a *process (thread) per message* approach with no explicit mechanisms or requirements for concurrency control. Event handlers of microprotocols are executed atomically in regards to concurrency, that is, one handler is executed after the other ([WHS01]). Composite Protocols may contain a shared state for which the programmer has to also ensure concurrency control.

Appia

Appia is “a protocol kernel that offers a clean and elegant way for the application to express inter-channel constraints.” ([MPR01]). It extends on many of the design concepts found in Ensemble [Hay98] and is entirely Java based. Appia is event driven and *Events* are object oriented data structures where new events can be derived by extending previously defined event classes. In Appia a *Layer* is the representative of a microprotocol. Microprotocols exchange information using *Events*. A S*ession* is an instance of a microprotocol. *Sessions* only interact with the environment through events. Layers define three sets of Events that are used to provide run-time compatibility checks. These are: 1) the Events it generates, 2) the Events it requires to provide its functionality and, 3) the Events it accepts.

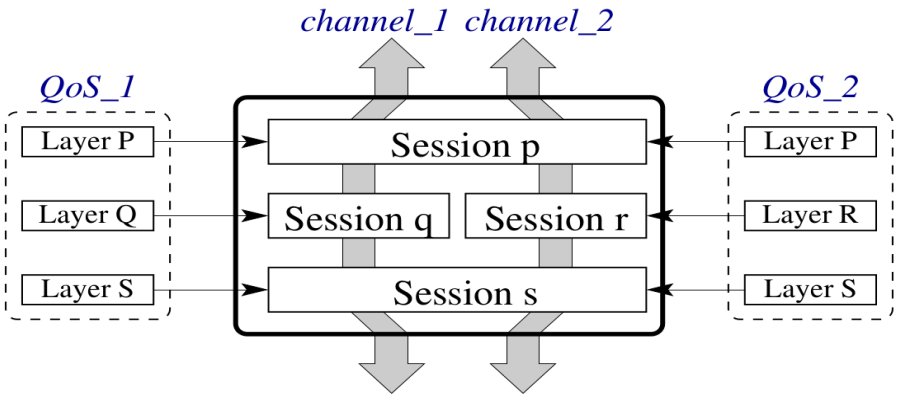

Fig. 11: Composition example in Appia: Stack with two channels. Both channels share sessions *p* and *s*. [Men06]

A *QoS* is defined as a stack of *Layers* and it specifies which protocols must act on a message and the order that it traverses. *Events* exchanged between two *Sessions* are delivered in FIFO order. *Channels* are instantiations of a QoS and are characterized by a stack of *Sessions* of the corresponding *Layers* ([MPR01]). Through implicit binding it is possible to associate specific *Sessions* to specific *Channels*. Appia also provides automatic binding and both can be used together. *Sessions* can be shared among different *Channels* (such as a failure detector Session) and they can be channel-aware through using a channel identifier that is delivered with every event. *Sessions* may use threads internally and Appia provides a special thread-safe method for generating events outside event handler invocations of that *Session*. By default, Appia uses a one-to-one mapping between event schedulers and channels and single threaded execution of all event schedulers but can be extended to support other models.

JGroups

JGroups (formerly called JavaGroups) is a group communication system that is also modelled after Ensemble and is written entirely in Java. Owed to the common design template both Appia and JGroups share many similarities. JGroups is event-based and strictly layered where each protocol represents a layer in the stack. Events can travel in both directions and have to be passed from one protocol layer to the next in FIFO order. Central to its architecture is the concept of *channels* where

channels with the same name form a process group ([Ban98a]). A channel can be understood of as a programming abstraction similar to a *socket.* Another essential concept are *building blocks* that provide pre-fabricated implementations of recurring design *patterns* and can be used on top of channels to extend their functionality ([Ban98b]). The following Figure (Fig. 12) outlines the JGroups architecture with a range of different building blocks that can be used and combined on top of a channel.

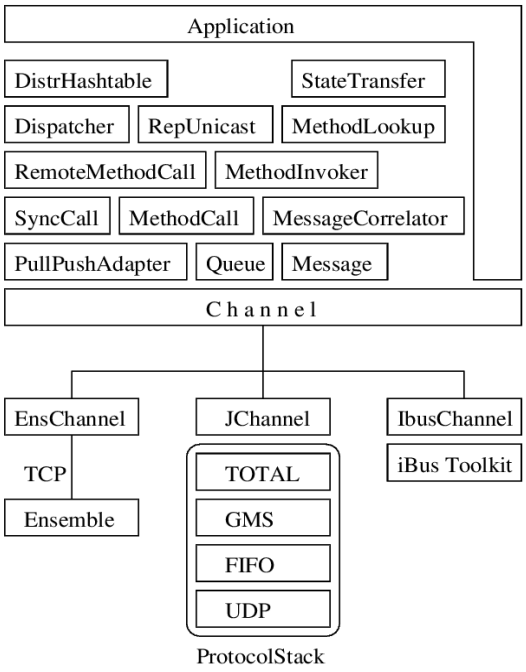

Fig. 12: Overview of the JavaGroups architecture [Ban98a]

Eva

Eva is a Java-based event driven protocol composition framework developed to build a group communication service based on a generic agreement framework ([BGH+01],[GHRT01]). The authors present an alternative approach to the layering strategy used in most modular group communication systems and protocol composition frameworks. Layering imposes many restrictions and can incur performance penalties because events and messages have to travel through the entire stack. Optimizations such as allowing events to bypass layers are far from flexible, often specific to a composition and also introduce additional complexity. ([BGH+01]) propose an object-oriented architecture based on the *Information Bus* abstraction ([OPSS94]). A number of cooperating objects are connected to each other via *event channels* to implement the given functionality. Objects can be *consumers* or *producers* of events and *event channels* are used to route events from a producer to consumers registered on that channel.

"*Applications built on top of EVA are composed of a set of cooperating objects, named components*, that communicate with each other via the exchange of events. A particular component is itself formed by cooperating objects, named entities, which may also communicate via the exchange of events." ([BGH+01])

Bast

Bast is an "open object-oriented framework for building reliable distributed application and middleware." ([GG97a]). It is based on protocols as basic structuring components and protocols are organized into a single inheritance hierarchy which follows protocol dependencies. Protocol objects can execute any protocol inherited from its superclasses and can run several executions of identical and/or distinct protocols concurrently ([GG97b]). A key element of the framework however is the use of the s*trategy design pattern* in a *recursive* manner to describe concrete protocols algorithms. The strategy design pattern's intent is to

"Define a family of algorithms, encapsulate each one, and make them interchangeable" ([GHJV94]).

Loosely speaking, the strategy pattern offers an alternative to subclassing where different *strategy objects* implement an *Interface* that represents a generalized algorithmic behaviour with a concrete implementation. In the context of Bast these *strategy objects* represent implementations of a distributed algorithm for a specific type of protocol (such as a concrete implementation for reliable multicast). Different protocol types are modelled by *protocol objects* that encapsulate such *protocol algorithms*. This means that the algorithms implementing a protocol are manipulated as separate objects and these *algorithm objects* execute *within protocol objects*. Bast employs the *strategy design pattern* because:

"... inheritance is not appropriate when it comes to choosing among several protocol algorithms *at runtime*. It is those limitations that lead us to *objectify* distributed algorithms by recursively applying the Strategy design pattern." ([GG97a]).

Generally it is not easy to plug in new state-of-the-art algorithms into existing group communication systems because they are a) too coarse grained and b) often monolithic in design. Using the strategy pattern is an approach to alleviate this issue and render the composition framework more flexible.

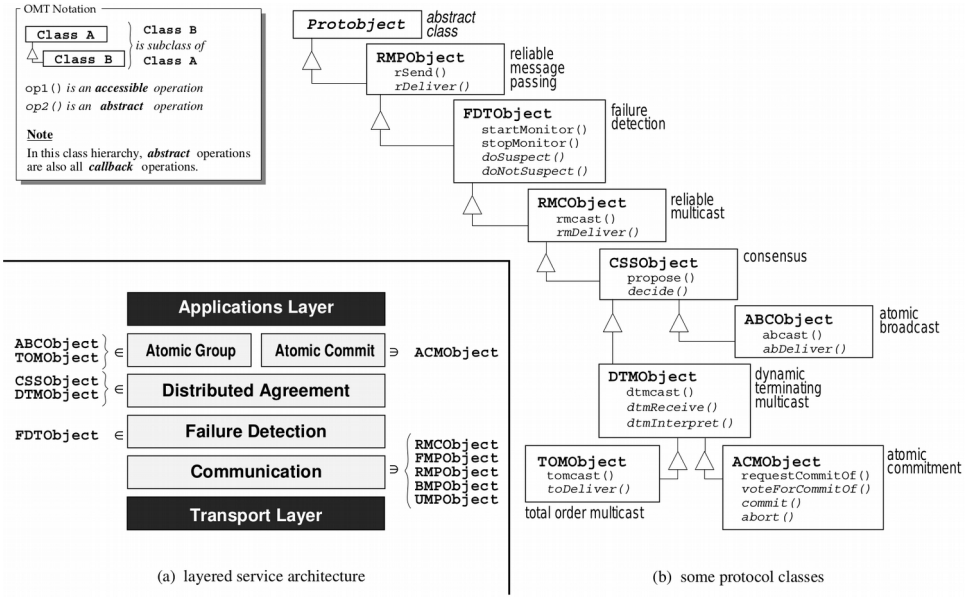

Fig. 13: Overview of the Bast framework [GG97]

Neko

Neko is a Java-based framework designed to support all phases of the development of distributed algorithms through a uniform and extensible environment ([USD01]). Distributed algorithms implemented with the platform can be tested both in a simulated environment as well as being deployed on real networks without requiring code modifications. This feature is achieved by providing a custom implementation for *Threads* and a network abstraction layer that behave accordingly to the execution mode (either as part of a discrete event simulation or the regular execution environment). Differences are hidden behind a uniform API that also allows the integration of other simulation models and execution modes without requiring changes in the actual protocols. By allowing the same code to be used for both discrete event simulations as well as in a regular manner development time is reduced. Neko provides a layered approach to protocol development with both *passive layers* and *active layers* that can own threads allowing for concurrent execution. Communication between layers is done through messages which are sent to lower layers using an asynchronous *send* operation and passed up the hierarchy using the synchronous method *deliver*. This stack-based approach is however not enforced and developers are free to use different structures, which is why Neko uses the term *microprotocol* for its protocols to reflect this aspect found in *microprotocol frameworks* ([USD01]). Protocols resolve dependencies on other microprotocols and components through *dependency injection*. Neko includes an extensive library of distributed algorithms such as different failure detectors, *atomic broadcast*, variants of *Consensus*, and other generic components that tie them together. These components were developed in the context of fault tolerant group communication and are to form the basis of a future group communication toolkit ([USD01]).

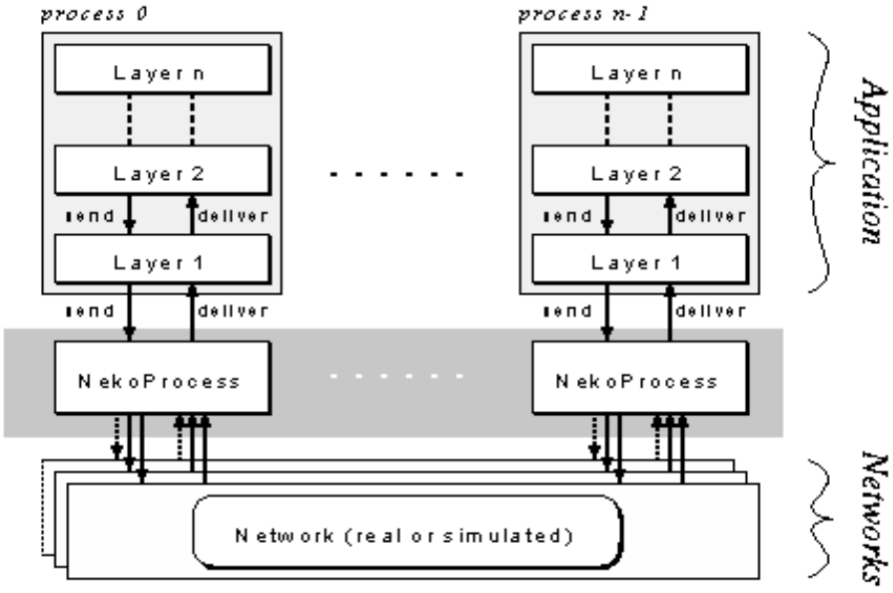

Fig. 14: Architecture of Neko [USD01]

Kompics

Kompics is a message-passing component model for building distributed systems developed to simplify the implementation of distributed algorithms and systems and make it easy to test and debug them. ([AH08],[Ara13]) Its underlying design uses *Components, Events, Channels, Ports, Event Handlers* and *Subscriptions* and provides a composition model that allows *nested hierarchical* composition and dynamic reconfiguration ([Ara13]). *Components* can be connected through bidirectional FIFO *channels* on *complementary ports* of the same (class) type. Each component specifies a set of ports it *requires* and *provides.* A port can be seen as as a service or protocol abstraction with an event-based interface that allows a specific set of event types to pass in each direction. An *Event* is a typed object having any number of typed attributes. *Event Handlers* accept events of a particular type or subtype thereof and are executed reactively on the reception of these events. Kompics allows for the decoupling of component code from the executor to allow for different execution modes including deterministic repeatable simulations (If no additional threads are created within components). *Handlers* of a specific component instance are executed mutually exclusively to simplify the design by avoiding the need for synchronization primitives on the instance's state. *Event Handlers* are *subscribed* to a specific port which needs to contain the handler's specified event or subtypes thereof. Dynamic reconfiguration can be achieved by first putting a channel on hold, which buffers all occurring events, and then *unplugging* the end or ends of the channel and *plugging* them in to the replacement components. After the relevant state of all the to-be-replaced components has been transferred to their replacement counterparts the channel is then *resumed* and operation continues with the replaced components. *Eventfilters* are also available for each direction of a channel in order to selectively filter the events passed onwards. This is for instance required when multiple (virtual) nodes share a network component in a local simulation, so that each node only receives the message events intended for them. Kompics provides an abstraction for blocking calls through a special *expect* primitive. Essentially a one-time handler for a specific event pattern is installed and once the event queue contains such an event it is removed, executed and the handler removed. Other events still remain in the queue so this breaks the FIFO ordering and can potentially cause programmers to introduce deadlock situations. The message-passing component model of Kompics in combination with the deterministic execution of handlers also allows users to quickly and easily create simulations. Components such as is used to describe the lower level networking functionality can easily be replaced by emulated counterparts, leaving a large portion of the original code and functionality unaltered.

Kompics was designed with usability and simplicity in mind and explicitly mentions its potential use case in an educational scenario, giving examples of how coursework was designed based on the framework.

Service Interfaces

In ([RWS06]) Rütti et al. argue that an approach based on *service abstraction* is more suitable for expressing modular protocols than the traditional *event based* abstractions. *Service Interfaces* provide several advantages such as adequate representations of protocol module interactions, fairly straightforward compositions of protocol modules, and integrated mechanisms to facilitate the implementation of *Dynamic Protocol Update* managers ([Rüt09]). *Protocol Module Interactions* are defined as follows ([RWS06]):

• *Requests* are issued by protocol modules. A request by a protocol module P_i is an asynchronous call by P_i of another protocol module.

• *Replies* are the results of a request. A single request can generate several replies. Only protocol modules belonging to the same protocol as the module that has issued the request are concerned by the corresponding replies. For example, a request by P_i generates replies that concern only protocol modules P_j

• *Notifications* can be used by a protocol module to inform (possibly many) protocol modules in the same stack about the occurrence of a specific event. Notifications may also be the results of a request.

In the *service-based* framework protocol modules in the same stack communicate through objects called S*ervice Interfaces*, where all of the above protocol module interactions are issued to them. A *Protocol Module* consists of a set of E*xecuters, Listeners* and I*nterceptors* that can be dynamically bound and unbound to S*ervice Interfaces*. Requests to a Service Interface lead to the execution of the bound Executor or are delayed until an Executor is bound. At most one Executor may be bound to a particular Service Interface. Listeners bound to a Service Interface handle *Replies* and *Notifications*. A *Notification* is handled by all listeners bound while a *Reply* is handled by one Listener which was explicitly specified in the corresponding *Request.* If the specified Listener for a Request does not exist the Reply is delayed until that Listener is bound. *Interceptors* that are bound to a Service Interface intercept the Requests, Replies and Notifications issued to that particular interface and can be seen as an Executor plus Listener. Protocol modules that have an Interceptor bound to a Service Interface are able to modify Requests, Replies and Notifications issued to that interface. If several Interceptors are bound to the same interface they are executed in the order of binding for Requests and reverse order for Replies ([RWS06]). The advantages of Service Interfaces over event-based abstractions are summarized in (Fig. 15). The described service-interface based approach has been implemented in an experimental protocol framework called SAMOA[29].

[29]In [WRS04] SAMOA is presented as “a programming language for a Synchronisation Augmented Microprotocol Approach” that is “... designed to allow concurrent protocols to be expressed without explicit low-level synchronization, thus making programming easier and less error-prone.” The therein presented framework employs an event-based approach and the Java implementation is called J-SAMOA.

	service-interface-based	event-based
Protocol Module Interaction	an adequate representation	an inadequate representation
Protocol Module Composition	clear and safe	complex and error-prone
Implementation of DPU Managers	an integrated mechanism	*ad-hoc* solutions

Fig. 15: Service-Interface-based *vs.* event-based. [Rüt09]

Header-Driven Protocol Composition

Using headers to route messages to the correct protocol in a stack is by no means a new method used for protocol composition and can be found in early works such as the *x*-kernel ([HP91]). The header based approach presented by Bünzli et al. ([BMN05]) however treats headers conceptually similar to remote method invocations and is inspired by functional programming. In the event-driven model *handlers* are bound to *events*, and they are executed when the event occurs. Generally many handlers can be bound to a single event, providing a *one-to-many* interaction scheme. For complex, event-driven compositions such a one to many interaction scheme can introduce compositional problems by mixing up the targets to which data should be delivered. Event based composition usually strives to *decouple* different components, however ([BMN05]) argue that modular decomposition of a complex protocol actually results in *tightly coupled* components. Tight coupling is meant in the sense that the role and properties of the sub-protocols are clearly defined, and that this tight connection is directed, where higher level protocols rely on precise lower level services ([BMN05]). The header-driven model can solve the compositional problems of the event model, simplify inter-protocol dependencies, concisely handle peer interactions and explicitly reveal their logical structure, and provide better static typing ([BFM+06]). In the header-driven approach it is assumed that for each *header* there exists exactly one matching *header handler*, *headers* replace events, *message dispatch* replaces event triggering and the binding mechanism of the event-based model is dropped. (because each handler will have exactly one *named* header matching it). A message can be seen as a sequence of headers, each having been added to the message as it traverses a path through the protocol stack. On the peer nodes, the message should take the reverse path defined by the sequence of headers, visiting each protocol in the correct order. Messages can “automatically” be routed to the right handler without requiring explicit upwards binding, simply by looking up the corresponding handler of the header. Because header-handler pairs are unique, the header type will always match. If protocol stacks are symmetric it can be guaranteed that remote message deconstruction will not fail and there are no type mismatches. In a sense, the header-driven approach can be seen as a deferred remote method invocation where each header invokes a remote method providing the required parameters. The question of concurrency in the header-driven model is not addressed and considered orthogonal to that of composition ([BMN05]) and updateable protocols are also not considered ([Men06]).

2.4 Simulation and Testing of Group Communication Systems

2.4.1 Testing Based on Reconstructed Global Executions

Verifying the correctness of a group communication system or components thereof can be a difficult endeavour. Elements such as *Consensus* in themselves present complex problems that require thorough testing to ensure that an implementation indeed meets the theoretical specification. When observed and tested individually a component might appear correct only to reveal faulty behaviour once it interacts with other components. Farchi et al. describe the situation as follows:

“Unfortunately, group-communication based systems are extremely hard to test and debug due to a number of stateful complex algorithms deployed in parallel and the unique combination of distributed and concurrent programming paradigms that amplifies the non-determinism in the system behavior.” ([FKK+05]).

Their approach towards assisting the testing and debugging of group communication based systems introduces a special *tester layer* that sits below the application layer in the protocol stack and monitors all message and event traffic at every node. The testing suite focuses on verifying *local* and *distributed invariants* ([FKK+05]) where, rather than creating distributed snapshots for later analysis, *distributed invariants* are checked by reconstructing a global execution from the individual log entries at different processes.

“While local invariants may be verified by looking at the local execution of each individual process, preservation of distributed execution invariants may only be confirmed by cross-matching executions of different processes in the system.” [FKK+05]

Timing differences between individual processes need to be taken into account as well and invalid or contradictory data has to be pruned when merging different execution traces to a consistent global execution.

“In order to verify these invariants, we need to correlate and analyze events that occur at different processes. … post-mortem log analysis is done based on the information that the Tester layer records into the log file. While still non-trivial, the analysis is facilitated by the fact that group communication systems inherently use logical timestamps for certain events.” [FKK+05]

It is outlined that the amount of data created is infeasible to be processed by humans and through specifying automatically checked invariants developers are only presented with relevant information where the required guarantees were violated. Some of the challenges that need to be faced when debugging GCSs are described by Farchi et al. as ([FKK+05]):

- The log files that are generated are large and distributed over multiple machines. It is hard to link events between different logs and the resulting data is infeasible to be analyzed by humans
- Deadlock detection is difficult and might not clearly show when analyzing log files

- Defining correctness criteria can be difficult for group communication systems where properties may be elusive or only “best effort”[30]
- Message ordering and safety guarantees can be checked by specifying tests on logged messages. It would be desirable to perform such checks for all test scenarios and not just those specifically designed for message ordering and safety, however the large amount of data created by logging all messages can be prohibitive.

The testing approach presented by Farchi et al. can help to analyse and debug GCSs that are executed in realistic scenarios where instances are deployed on real or scaled down test systems. However distributed systems testbeds can be costly and time consuming to set up and the approach is less feasible for earlier prototyping, hence an obvious approach to mitigate these issues is to simulate parts of or even the entire distributed system locally ([CBCP11]). Simulating a distributed execution and testing based on the reconstruction of a global execution are methods for testing that complement each other.

When creating a simulation environment for GCSs the fundamental question arises on how much of the real system should be abstracted away. Generally speaking, the more abstract a simulation model becomes, the more adaptation is later on required if the code is to be executed in a real environment. This obviously directly influences the results obtained, especially in the context of performance metrics. Complex concurrent behaviour or issues particular to a specific implementation can be unintentionally hidden away through the simulation model. Unless the composition framework specifically supports switching between simulation and regular execution such as Neko ([USD01]), algorithms will have to be implemented twice: once for the simulation and then again for the actual target system. On the other hand the level of abstraction afforded by the simulation can make the initial development process less complex and error prone and place more focus on the formal correctness of the algorithm and provided guarantees of the GCS. Both approaches have their merits and ideally one would progress from a highly abstract simulation towards an increasingly realistic scenario.

The following works specifically address the simulation and testing of group communication systems or complex distributed algorithms and protocols. They are presented starting with more abstract simulation environments, progressing towards more complex and realistic simulations.

2.4.2 Efficient Simulation of View Synchrony

In ([DH12a]) Drejhammar and Haridi present an algorithm for efficiently simulating view synchrony, including failure-atomic total-order multicast, for the partitionable group membership model in a discrete-time event simulator. The removal of requirements for third party middleware and detailed network simulation greatly reduces the complexity of the environment. Basically the approach takes advantage of the ability of the discrete-time event simulator to have knowledge of and be able to modify the global state of the system. It can can hence ensure virtual synchrony guarantees for all processes simply by re-ordering events and messages according to the provided virtual synchrony specifications. Beyond the required guarantees messages and events also experience random delays so that they (probabilistically) express all permissible timing behaviours

[30]See chapter 2.2.6 The Group Membership Problem and Virtual Synchrony and [CKV01] why properties are often only best effort.

of the specification. The model is presented for partitionable group membership but can readily be adapted for the primary-component model. Simulation of the network is done in an abstract manner and provides reliable message delivery and atomic broadcast capabilities to correct nodes within partitions. Because implementation specifics are removed the API is concise and reminiscent of the generalized and abstract specification of virtual synchrony layers found in works such as ([CKV01]).

2.4.3 Simmcast

Another approach at the simulation of multicast protocols at a more abstract level is presented by Muhammad et al. with the *Simmcast* framework ([MB02]). This process-based discrete event simulator written in the Java language provides abstract building blocks that need to be extended to form concrete implementations of simulation scenarios. The authors describe their work as:

“Simmcast approaches network simulation from a different perspective. While in general-purpose network simulators the new protocol is added to the existing model, in Simmcast the model is built along with the protocol, through instantiation of the framework. This bring clear advantages in terms of flexibility. Simmcast can be thought of as a network-research-oriented simulation engine with built-in support for group communication/distributed systems facilities.” ([MB02])

In the context of ([MB02]) the term *group communication* is used as a more generalized description that also includes unreliable multicast protocols for addressing dynamic or static groups such as IP-Multicast. The provided group abstraction of the simulator nevertheless gives a consistent view of the membership where join and leave operations are instantaneous. View synchronous group abstractions are not included but the simulation model can be extended to provide such functionality, for instance by using an approach such as that of ([DH12a]). Simmcast employs dynamic class loading that allows for a configuration at runtime. The APIs for many simulation components such as network sockets are closely related to the real Java APIs with the intention of simplifying the adaptation of simulation code for an execution in a real environment. Trace file generation is flexible and can readily be extended to produce compatible output for visualization tools ([MB03]) such as the *ns-2* network animator *nam* ([Meh01]).

2.4.4 Neko

The *Neko* framework has already been covered in chapter 2.3.2 Protocol Composition Frameworks for GCSs so here the focus is placed on its simulation aspects. Neko addresses the problem that both testing and deployment in real scenarios generally require different implementations of a distributed algorithm. While Neko's focus lies on performance engineering the observations hold up for the development of distributed algorithms in general. Neko presents the programmer with a protocol composition framework where certain libraries and classes such as the lower networking component and *Threads* (in this case a special *NekoThread* class is provided with a similar API to regular Java Threads) can be configured to execute in both a regular fashion or as simulated components using Neko's own discrete event simulation engine. These libraries are implemented using the *strategy pattern* as to facilitate the easy addition of new execution modes such as alternative simulation engines ([USD01]). As long as programmers adhere to certain guidelines such as avoiding *static* (e.g global) variables and use these provided libraries rather than language primitives, the switching between simulation and real execution becomes straightforward. The desired execution mode merely

needs to be specified within a configuration file. Neko is focussed on the application layer, rather than the simulation of complex lower level networking aspects ([USD01]), but allows for more detailed network simulators to be integrated if more realistic simulations are required.

2.4.5 MINHA

MINHA is a Java-based (distributed middleware) simulation environment described by its authors as:

"a system that virtualizes multiple JVM instances within a single JVM while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of the actual distributed system." ([CBCP11]).

A key aspect of the system is the (transparent) transformation of language primitives such as concurrency, I/O and networking from regular code to simulated counterparts. Rather than providing specialized libraries with their own API, MINHA employs *bytecode instrumentation* to convert regular Java code so its execution is synchronized with respect to the discrete event simulation of the simulated components. Furthermore the execution of non-simulated code segments is timed and reflected in simulation time to provide a more accurate representation of timing and performance measures. The hereby created simulation is well suited for testing distributed services as it can cover more of the complex behaviours and interactions that may not be reproduced with simpler simulation models and tests. MINHA can be used to validate actual implementations of middleware or distributed services with complex, concurrent software components whereas other simulation models might only validate the design. Carvalho et al. ([CBCP11]) describe the contributions given by MINHA as:

- "Virtualizes a significant portion of modern Java, allowing off-the-shelf code to run unchanged, including threading, concurrency control, and networking. In fact, it provides something akin to a Java hosted hypervisor, transparently running multiple "virtual" JVMs within a single host JVM."
- "Provides simulation models of networking primitives and an automatic calibrator, that adjusts model parameters such that it mimics an existing hardware/operating system combination. This allows performance results obtained with MINHA to be compared with a real system."

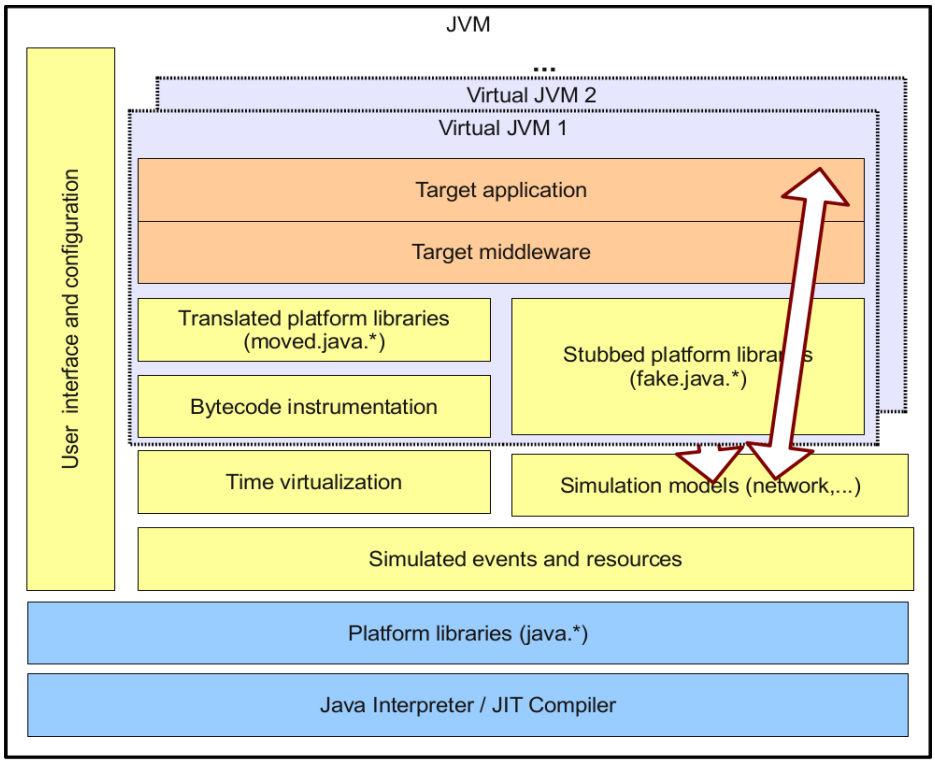

Fig. 16: Simulation of a distributed Java application [CBCP11]

2.4.6 Simulation Environments not explicitly designed for GCSs

Aside from the presented works specifically focused on simulating GCSs and complex protocol stacks there also exist a wide variety of (lower level) network simulators such as *Omnet++* ([Var01]) or the *ns-2 network simulator* ([MFF+97]) and other specialized simulation environments ([CKP+04],[BWWZ05],[BHvR05],[TADM06]) that can also serve as a basis for simulating and testing GCSs. Network simulators are generally better suited for later stages of development where realistic test scenarios and performance metrics are of interest. Modifying GCSs and protocol stacks to be compatible to the (discrete event) simulators that are used in these simulation environments can however be problematic. Timing assumptions need to be consistent for both the simulated components and regular code or else at best results will be skewed and at worst the simulation itself may introduce faulty behaviour. Either an approach similar to that of Neko ([USD01]) can be chosen where the protocol composition framework or execution environment is designed to provide a common API for both real and simulated execution or *emulation* is used in combination with regular code. The former approach requires that the GCS or protocol stack is specifically designed to use the provided API whereas in the latter case code is not tied to a particular API. In ([Fall99]) Fall discerns between two different paths of emulation*: environment emulation* attempts to recreate the execution environment of the protocols allowing them to be executed without any modification in the simulator where as *network emulation* aims to allow simulated components to communicate with protocol implementations in the real-world. Rather than having to simulate the entire environment, network emulation only requires that the to-be emulated library methods are intercepted or otherwise replaced in the protocol stack. This approach renders the simulation less complex, but needs a real-time capable simulator and also introduces other issues such as the potential for a *livelock* ([Fall99]).

Weingärtner et al. ([WSHW08],[WRSW10],[WSvL11]) propose *synchronized network emulation* using virtualized hosts as a solution for some of the issues encountered in *network emulation*. Their approach employs a synchronizer for both the execution of the simulator as well as the virtual machines used to run the target protocol(s) to avoid the real-time requirements of *network emulation*. Virtualized hosts and the network simulator execute within time-slices and halt once they reach the time boundary until the synchronizer assigns them a new slice once all components have reached the boundary. The presented method is similar to *environment emulation* but leverages the advances in (hardware) virtualization to avoid having to use a complex simulation of the execution environment. *AgentJ* ([TADM06]) and the previously mentioned *MINHA* ([CBCP11]) are also examples where forms of synchronized network emulation are used. Other simulation environments in the general topic area of distributed systems and networking that are of particular interest to this work are *DAP* ([CKP+04]) and *SANS* ([BWWZ05]). Both works explicitly mention their suitability for an educational context and also provide visualization elements for their simulations.

DAP (Distributed Algorithms Platform)

“DAP (Distributed Algorithms Platform) is a generic and homogeneous simulation environment aiming at the implementation, simulation, and testing of distributed algorithms for wired and wireless networks.” ([CKP+04]).

DAP's goals are, amongst others, to provide flexible environment creation while allowing programmers to use the c++ programming language in a real-world scenario like setting. It is explicitly described as a “tool for experimental analysis and for demonstration or educational purposes” ([CKP+04]). The design of DAP is similar to Neko ([USD01]) in that it uses abstractions of processes and networking elements that can be configured for the execution in a real environment (in the case of DAP the code is linked with provided libraries to execute in a real environment). These abstractions are structured into three different layers, namely the *Topology Layer*, *Computing Layer*, and *Process Layer*, that are described as follows ([CKP+04]):

- *Topology Layer***:** It describes the location properties of the computing nodes. It includes topology nodes that specify positions and topology links that specify connections between these positions.
- *Computing Layer:* It describes the properties of the computing nodes. Each computing node resides on a topology node.
- *Process Layer.* It represents the processes that make up the simulated algorithm. Each process executes on a computing node. The processes may modify characteristics of the lower layers (e.g., close a communications channel or move the computing node from one topology node to another).

DAP also includes a *Graphical User Interface* (GUI) on the basis of LEDA ([MN99]) for the construction and real time monitoring of simulations. Since communication with the simulation engine is based on TCP/IP it is also possible to implement a GUI in the language of choice if the communication protocol is followed. DAP's GUI enables the monitoring of most aspects of the simulation in real-time while also allowing pausing and modification to elements of the simulation during execution. The GUI also provides modules for designing the topology, constructing scenarios,

and recording and viewing statistics. An interesting aspect of DAP are its scenarios. They can be constructed to include stochastic elements such as failure probabilities as well as specifically defined events such as interrupts or node failures at a specified simulation time. The simulation itself can be distributed and the GUI allows for a passive and active role which consists of one controller who can actively influence the simulation and a number of passive observers. DAP focuses mainly on algorithms for wired or mobile wireless networks and not on complex protocol stacks. Implementing stacks is possible if each layer is implemented as an individual process on a computing node.

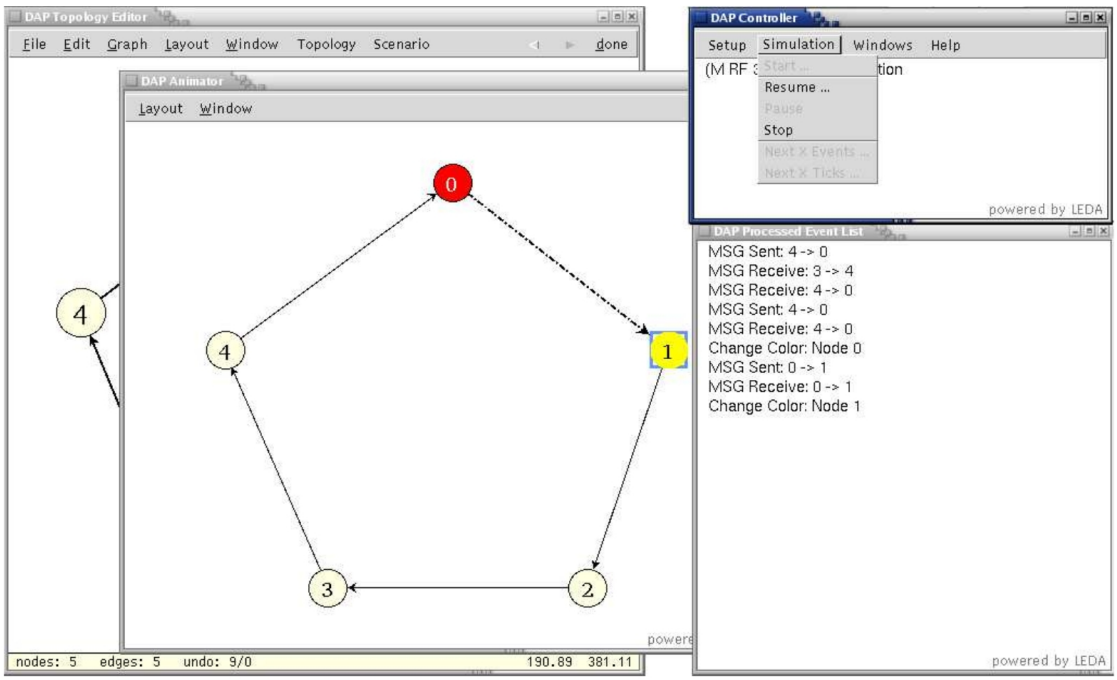

Fig. 17: DAP GUI windows while executing an algorithm [CKP+04]

SANS (Simple Ad Hoc Network Simulator)

SANS ([BWWZ05]) is a *network emulator* particularly intended for developing distributed programs and protocols on the networking and transport layer by emulating physical and data link layers through a custom implementation of Java UDP (User Datagram Protocol) multicast. It is targeted towards wireless networks and the UDP multicast interface was chosen because of its one to many interaction scheme that shares similarities with radio communication. As an added benefit using the UDP interface allows developed code to be easily transferred to real-life systems. An interesting aspect in regards to the goals of this work is that SANS was developed with the goal of incurring very little learning overhead and also provides a *Graphical User Interface* (GUI) to assist the setup of simulation scenarios. Burri et al. summarize the key features of the system offers as follows: ([BWWZ05])

- Ease of use
- graphical representation of the network
- real time network simulation
- code developed on the simulator should run on “real-world” hardware without adaptation
- individual transmission ranges for each node
- adjustable link properties as delay and packet loss
- support of a standard programming language

SANS also allows changes to the network topology as well as the addition or removal of nodes at runtime. Configured network topologies can also be saved and loaded. The simulation framework as well as the client programs all execute within a common JVM. To avoid that different nodes share a common *namespace* and could hence leak information through global variables a custom Java *ClassLoader* is used that generates a unique namespace for each node. Clearly, as SANS is a *network emulator,* the previously discussed issues in regards to real-time execution also apply. Furthermore the visualization is directly tied to the execution and hence observing the progress of the simulation run can be problematic if the protocol isn't artificially slowed down.

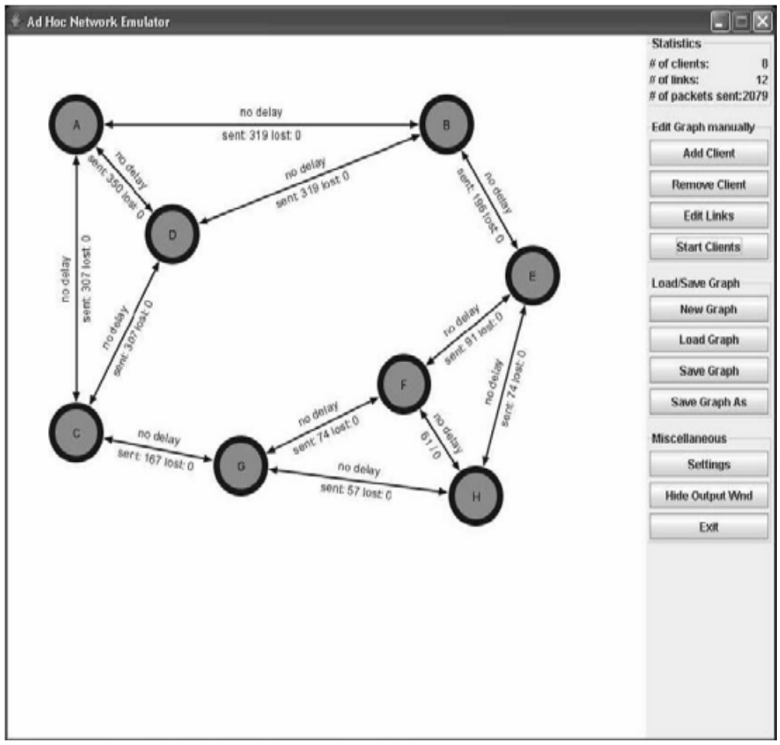

Fig. 18: SANS main window showing a network consisting of eight nodes which are partially connected. [BWWZ05]

2.4.7 Summary

The presented simulation environments for group communication systems and distributed algorithms provide different advantages and disadvantages. Generally, more abstract simulations are less complex in their design and are best suited for prototyping new algorithms and approaches and testing their correctness. The performance results obtained through such simulations are indicative of the general runtime characteristics to be expected of a real implementation using the particular design. The API of abstract simulation environments is usually not representative of actual programming language primitives and so portability of implementations is hindered. More detailed simulation environments strive to capture the complex interactions found in implementations of group communication systems and complex protocol stacks. While such simulation environments can also be used to verify the general correctness and characteristics of distributed algorithms and protocols, they are better suited for performance analysis and optimization as well as uncovering implementation specific problems. This can range from concurrency issues such as deadlocks or thread starvation, timing problems, to ensuring correct behaviour under a wide range of external inputs and failures. To be able to provide this functionality complex simulation environments need to allow the execution of a large part of the unmodified protocol code and its use of language and library primitives. In this case timing becomes an issue as the execution of “regular” code and simulated components need to be synchronized in respect to each other. Simulation environments targeted at GCSs generally do not place a strong focus on their ease of use and appropriateness for an educational context. Examples of how such tools may be augmented to address these issues can be found looking at a broader range of network simulators. To gain a deeper insight into the requirements for an educational simulation environment the next chapter (2.5) covers works explicitly designed for that context.

Testing the correctness of a group communication stack or other complex protocol compositions can be a difficult endeavour. In this regard the flexibility offered by modern composition frameworks and modular GCSs renders the process even harder, as it is the interaction between individual components that defines the overall characteristics of the system. Replacing even just one protocol layer from a composition may invalidate any previously obtained test results. Farchi et al. ([FKK+05]) describe an approach where *invariants* based on a global reconstruction of the distributed execution are used to verify the correctness of a composition. This testing method can be used independently of whether the composition is executed in a simulated environment or during regular execution.

2.5 Educational Software Tools for Distributed Systems Topics

2.5.1 Overview

Available literature covering the design of educational simulation environments for advanced distributed systems and distributed systems topics in general is scarce in comparison to that of many other areas in the field of computer science. Wein et al. for instance point out that there are few innovative methodologies for teaching distributed systems compared to other topic areas ([WKC09]). Especially introductory level courses are covered extensively while there is significantly less information for more advanced topics. The author is not aware of an educational simulation environment that specifically addresses the field of interest, namely *group communication*. Hence the scope of the surveyed literature needs to be extended to include works from both the general field of distributed systems as well as other areas of ICT that appear relevant for developing an educational group communication system. Surveys of literature on teaching introductory programming such as the work presented by Pears et al. ([PSM+07]) can offer guidelines and starting points for designing not just introductory coursework, but also material for more advanced topics. In regards to (programming) tool development designed specifically for educational purposes they point out that amongst the large body of research and development only a small number of tools are used outside their home institutions. Pears et al. reason that this is because the tools are mostly designed for a specific local problem, often lack the ability to be easily modified and don't offer support over longer periods of time. This observation might also serve to explain the lack of availability of publications for educational tools for advanced distributed systems topics, as developed tools for a specific course may simply not be published to a wider audience. Approaches for designing coursework like those of Wein et al. ([WKC09]) and Mishra ([Mish02]) can also provide insights through describing the methods used in actual distributed systems courses they have conducted. Wein et al. use virtualization and *game-based learning* to provide students with a realistic scenario where they have to manage a distributed system while a game coordinator introduces failures and other problems to the system. (Hardware) virtualization allows for more realism in the course as larger distributed systems can be emulated, saving both setup time and costs involved in running actual hardware. Mishra emphasizes that assignments should not just be a collection of individual exercises where students have to use a certain technology or implement a specific algorithm for each individual exercise. Exercises should be constructed so that they add increasing complexity and functionality to a single, more general task. A model for structuring such assignments is presented that is illustrated via an example where a peer-to-peer look-up service for a distributed system is to be designed, implemented and evaluated.

The composition framework Kompics ([AH08],[Ara13], also see 2.3.2) may be the most closely related work in regards to the goals set out by this thesis. This protocol composition framework was specifically designed to simplify the creation and testing of distributed algorithms and complex protocol compositions. Potential uses as a teaching tool are described and its component model and support for deterministic execution allows for a relatively easy construction of test scenarios. Specifically, ([AH08]) mention that Kompics was used as an alternative to Appia ([MPR01]) in an advanced distributed systems course at the KTH Royal Institute of Technology, where students had

to implement various distributed abstractions as reactive components. He reports that students gave very good feedback regarding its ease of use. Kompics is not specifically designed as an educational tool but shares many of the goals found in educational works. It is mentioned in this section because of its stronger ties towards teaching than for instance Neko ([USD01]), which is arguably more focused towards simplifying protocol prototyping and testing for research purposes.

Although most papers on educational simulation frameworks and approaches for designing coursework for distributed systems do not explicitly mention the pedagogy on which they are based, they follow a similar design philosophy. The teaching methods and designs are often in accordance with aspects of *constructivist* learning theories and educational tools mostly employ some form of *algorithm visualization.*

Cognitive Constructivism

Using realistic problem scenarios where students are to engage in activities similarly to those encountered in "real life" situations is a proposition that is supported by *constructivist* learning theories. Basing computer science courses on constructivism has, for instance, been suggested by Ben-Ari ([Ben-A98]). Barg et al. ([BFG+00]) describe their experiences of using *Problem Based Learning* (PBL), an instructional model based on constructivist principles, which they applied to foundation courses in computer science. Cognitive constructivism "is a theory of learning which claims that students construct knowledge rather than merely receive and store knowledge transmitted by the teacher." ([Ben-A98]) and has become the dominant theory of learning ([Ben-A98]). The core concept of constructivism is that rather than separating what is learned from how it is learned, the understanding of an individual is seen as a function of the content, the context, the activity of the learner, and, perhaps most importantly, their goals ([SD95]). Constructivist learning theories suggest that active engagement rather than passive learning will lead to the construction of new knowledge.

Even if the underlying pedagogical theories are not explicitly mentioned by many approaches described in the literature[31], they usually follow the model of trying to provide students with realistic learning environments that closely match real situations they could encounter in their later professional careers (see for instance [Mish02, WKC09]). For these approaches the learner's tasks often adhere to many of the guidelines that are described for *Problem Based Learning* (PBL) environments such as those given in ([SD95]). It needs to be pointed out that PBL should not be misunderstood as merely giving students realistic exercises that are however still tightly defined.

"PBL involves much broader problems, which involve a larger set of problem solving skills. Critically, PBL places problem-solving and metacognitive skills at the heart of the curriculum." ([BFG+00])

Algorithm Visualization

Teaching in the field of distributed systems brings with it a number of challenges amongst which one is often cited as being especially difficult. That is, to make clear to students the concurrent nature of the execution of a distributed algorithm on a number of processes in the absence of a global clock ([Mish02],[O'Donn06],[BR01]). The dynamic nature of algorithms in general, and distributed algorithms executing concurrently on multiple nodes in particular, do not easily lend themselves to

[31]Here literature refers to descriptions of coursework and assignments for (advanced) distributed systems topics

be visualized in a static way such as using illustrations. Visualization of algorithmic behaviour has traditionally played an important role in the development of learning materials and has been employed in computer science education for a long time ([SCA+10]). Of the educational works reviewed for this thesis a majority contains some form of *algorithm visualization* (AV) element, however the reasoning for its inclusion is more often anecdotal or based on intuition rather than relying on specific learning theories or studies that show AV effectiveness and how it should be incorporated. Interestingly a meta-study conducted by Hundhausen et al. on the effectiveness of algorithm visualization revealed "that *how* students use AV technology has a greater impact on effectiveness than *what* AV technology shows them" ([HDS01]). It is a fundamentally important conclusion because the study suggests that even if an AV component produces correct visualizations it may provide little or no extra benefit to learners if designed incorrectly[32]. The by ([HDS01]) surveyed studies were categorized into four different learning theories upon which their assumptions for the effectiveness of algorithm visualization are based[33]. Of the different learning theories, namely *epistemic fidelity, dual-coding, individual differences*, and *constructivism*, they found that cognitive constructivism is the most robust to predict learning outcomes. As previously described, constructivism assumes that learner interaction and active engagement is a key element in the learning process. In an ITiCSE (Innovation and Technology in Computer Science Education) working group on improving the educational impact of algorithm visualization Naps et al. also advocate the thesis "that visualization technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity." ([NFM+03]). They present a new taxonomy of learner engagement with visualization technology and suggest metrics based on Bloom's taxonomy ([BK56]) for assessing learning outcomes that may arise from such engagement. AV is often encountered in tools designed for more basic topics whereas "upper-division" courses such as those covering NP-completeness get much less attention ([SCA+10]).

The educational simulation tools presented in the following section cover distributed algorithms and protocols in general and are mainly targeted towards simulating a single algorithm. They present most of the more recent works on the topic area. The author is not aware of educational works specifically focused on group communication or more complex protocol stacks.

[32]Namely, if the AV component does not offer any form of interactivity to learners and places them in a passive role
[33]Some studies actually fit into two categories

2.5.2 Educational Simulation Tools for Distributed Algorithms and Protocols

Lydian (Library of Distributed Algorithms and Animations)

An often cited framework in the context of educational simulation environments for distributed algorithms is LYDIAN ([KPT06]), developed at the Chalmers University of Technology. It is described by its authors as: “an environment to support the teaching and learning of distributed algorithms. It provides a collection of distributed algorithms as well as continuous animations.” ([KPT06). The framework is intended to serve a wide range of educational purposes ([Kol05]) with a focus on the visualized simulation of distributed algorithms. A key feature is the possibility for users to create their own network structures and protocols to be run on the simulator. The event driven simulation engine DIAS uses communicating finite state machines and is based on the work of DSS ([STP+92]). LYDIAN uses the concept of experiments to describe different scenarios. An experiment contains information about the protocol, the underlying network structure, the animation and the generated trace file during execution. Experiments can be simulated either with or without an animation. The creation of network descriptions is facilitated by a component based on LEDA ([MN99]) where users can simply draw networks based on a graph model in which vertices are processes and edges are connecting links. It is possible to specify the timing behaviour of the network. Protocols are written in an own language based on C syntax in which function calls are defined for state and event pairs. All possible events based on states and messages are listed in a table of transitions which is created interactively when a new protocol is created. The animation framework of LYDIAN is structured to provide a set of views to the user, taking into account the difficulties learners experience when trying to understand distributed algorithm execution caused by the absence of a global time ([Kol05]). These additional views differ from the basic view that is used to illustrate the fundamental concept of the algorithm in that they are transferable between different message passing protocols whereas the former is individually designed for each algorithm. The *communication view* is used to measure communication complexity through visualising processes' message contributions. The *causality view* illustrates the causal relation between events and also shows how logical clocks are incremented during execution ([Kol05]). This view is available even if an algorithm does not use logical clocks. The *process step view* provides information about a node's status at any point during the animation and also can be viewed interactively. The *process occupation view* shows the actual time each process is kept busy by the algorithm and is derived from the simulation trace. A case study was conducted ([KPT03]) on an undergraduate distributed systems course at Chalmers University of Technology where students were given the choice of implementing either leader election based on echo-broadcast, leader election based on voting or resource allocation based on logical clocks.

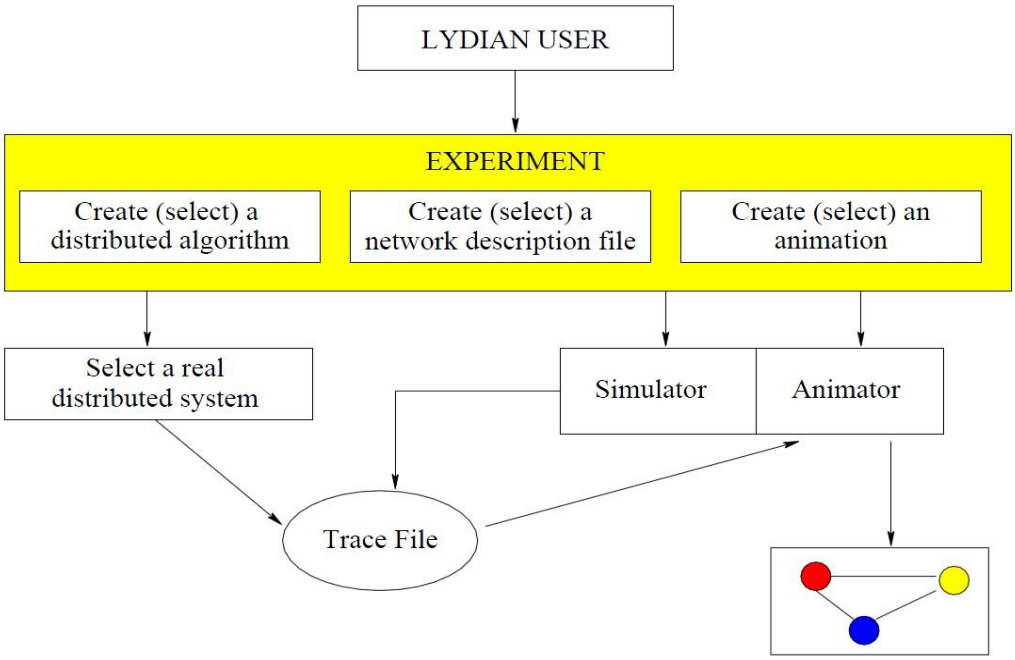

Fig. 19: An overview of LYDIAN's functionality from a users perspective. [Kol05]

DAJ (Distributed Algorithms in Java)

"DAJ (Distributed Algorithms in Java) is a framework for writing Java programs to implement distributed algorithms. The programs display the data structures at each node and enable the user to interactively construct scenarios." ([Ben-A01]).

This framework presented by Ben-Ari focusses on an educational use at high-school or undergraduate level aiming to teach a basic understanding of the presented algorithms and allowing students to interactively create and analyse scenarios. DAJ is Java based using the Swing library as the basis for its Graphical User Interface. The focus however lies not on animation or elaborate visualization but rather on being able to easily control the scenario and providing students with a "full data display" ([Ben-A01]). To facilitate this a single display is used which is divided into panels, each corresponding to one node participating in the simulation. This design decision poses limitations on the amount of nodes that can effectively be visualized[34]. The GUI provides buttons to interact with nodes and the overall simulation, allowing only relevant messages to be sent. The simulation model itself is based on message passing, however it is up to students to initiate steps:

"... instead, the student is required to select messages to send that will be compatible with the algorithmic characteristics of each node." ([Ben-A01]).

Algorithms are developed by implementing the abstract class *DistAlg* which is central to the framework. DistAlg also encapsulates algorithm-specific GUI components. Distributed algorithms should be implemented as state machines specifying actions and GUI options depending on the state and received message of a node. Ben-Ari summarizes the design of DAJ as ([Ben-A01]):

[34]This will depend on the screen size but Ben-Ari lists six nodes as a sensible upper limit

- All the information about the state of the distributed system is presented in one large window.
- The system makes discrete (untimed) state changes, and the student is responsible for selecting the state change at each step.
- The student will be supplied with prompts as to the allowable state changes.
- A logging facility enables the student to restart an algorithm and restore a previous state.
- The structure of the display is identical in the implementations of all the algorithms.
- DAJ is a portable Java program, which can be run either as an application or as an applet, though the logging facility can only be used in an application.
- The implementation of a new algorithm requires only general Java programming proficiency and no knowledge of the Swing GUI programming library.

To illustrate the functionality of the framework and its GUI a DAJ implementation of the Byzantine generals algorithm ([LSP82]) for four generals is used (Fig. 20).

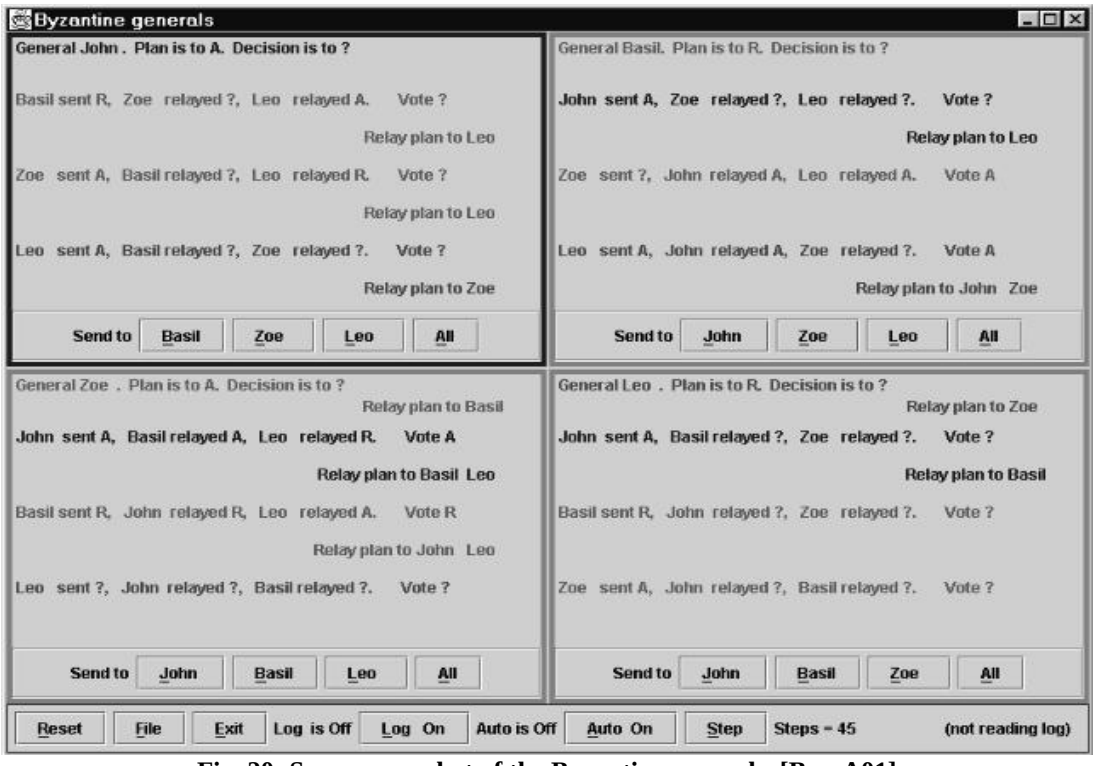

Fig. 20: Screen snapshot of the Byzantine generals. [Ben-A01]

HiSAP (Highly interactive Simulation of Algorithms and Protocols)

HiSAP is an architecture developed by Burger and Rothermel at the University of Stuttgart that consists of a framework to: "... build simulations and generate applets from formally specified algorithms or protocols." ([BR01]). The design aims at providing teachers or other experts a tool to easily develop interactive visualized simulations of distributed algorithms and protocols. It is argued

that in most cases it is much easier to specify a protocol in some specification language rather than having to write a new applet for its simulation. HiSAP consists of a simulation and animated visualization component where input is defined either through a script or directly by the user. Simulation models can be derived through the transformation of a formal specification, with the system currently providing a prototypical implementation for a special variant of Petri nets and plans for further methods such as supporting SDL (Specification Description Language) or PROMELA ([Hol91]). The predefined scripts are intended to for non-interactive simulations where key aspects of the protocol are outlined while direct interaction with the generated simulation applet is more interactive. While HiSAP assists in the creation of Java-Applets, parts still have to be implemented manually such as elements referring to the protocol logic and components of the visualization ([PB01]). The architecture was also designed to allow for extensions such as further tool integration or components to validate correctness or estimate performance. HiSAP is focussed towards simplifying the creation of interactive protocol simulations for experts or teachers rather than giving students the ability to experiment or test their own protocol implementations. This is reflected by the use of formal specification methods to generate simulations, where correctness and the abstract functionality of an algorithm is at the centre of attention. An evaluation of the educational effectiveness was conducted surveying students of three graduate courses after their utilisation of three different applets. The results were mixed with an applet describing the asymmetric two-way authentication protocol showing no obvious influence while the other two (token ring and adaptive synchronization protocol) were rated as helpful and “Only those students who had worked with the applet really understood the algorithm and protocol, and most answered that the applet had been crucial for their understanding.“ ([BR01]).

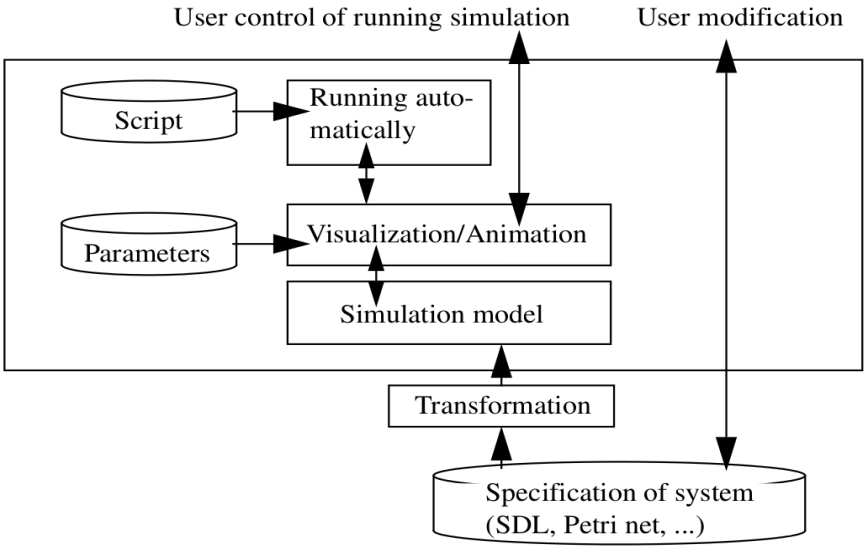

Fig. 21: Basic architecture of HiSAP. [BR01]

T-DAJ[35] (*Toolkit for* Distributed Algorithms in Java)

T-DAJ is a Java-based toolkit developed by W. Schreiner at the Research Institute for Symbolic Computation of the Johannes Kepler University in Linz, and is aimed at the visualization and development of distributed algorithms. The motivation stemmed from the "... lack of an easy to use and universally accessible platform for implementing the algorithms taught in class and investigating their dynamic behavior." ([Schr02]). T-DAJ's system model consists of a network of nodes partially linked by channels through which asynchronous and independent point to point messages are communicated. Nodes each have a local time which is implicitly advanced by communication operations and is accessible to the programmer. The modes of execution for the simulation are either as a standalone application with or without visualisation as well as the option to embed the simulation as a Java Applet in a web page. Each node is run on a separate Java Thread, returning control to the scheduler on message passing operations. Programs executing on nodes use an imperative message passing style which is more familiar to learners and closer to many real-world applications rather than modelling them as state machines. Users can define their own node schedulers to determine the next ready node or message selectors to determine the next message to be delivered. This allows for various execution and failure models such as synchronous network execution or duplicate message delivery to be introduced. Nodes are generated by calling a function with the to-be executed program, the node's label, and the coordinates for visualization passed as parameters. The user can hereby specify different programs for nodes. Defining assertions about the global state of the network provides the ability to check the validity of invariance conditions. A method *getText()* can be overridden for Message and Node classes to return a single line of text describing the internal state or content to be visualized. Nodes and channels are automatically visualized in an abstract graph representation. A colour coding scheme is used to communicate the different states of nodes and channels to the user such as red for blocked nodes waiting for a message or green for nodes that are ready for execution. Network execution speed of the simulation may be slowed, interrupted, stepped through or restarted and nodes can be repositioned via drag and drop. T-DAJ has been used in classrooms since 1997 mostly for exercise and project work and comes with a small library of algorithms that include ([Sch02]): distributed snapshots, parallel convex hull construction, termination detection, breadth first search, invitation algorithm, Maekawa's mutual exclusion algorithm, LyHudak mutual exclusion, Ricart-Agrawala mutual exclusion, Dijkstra-Scholten termination detection, and total order broadcasting

[35]To avoid confusion between the equally named tools of [Ben-A01] and [Schr02] the naming convention introduced in [Piy11] is hereby used in this work.

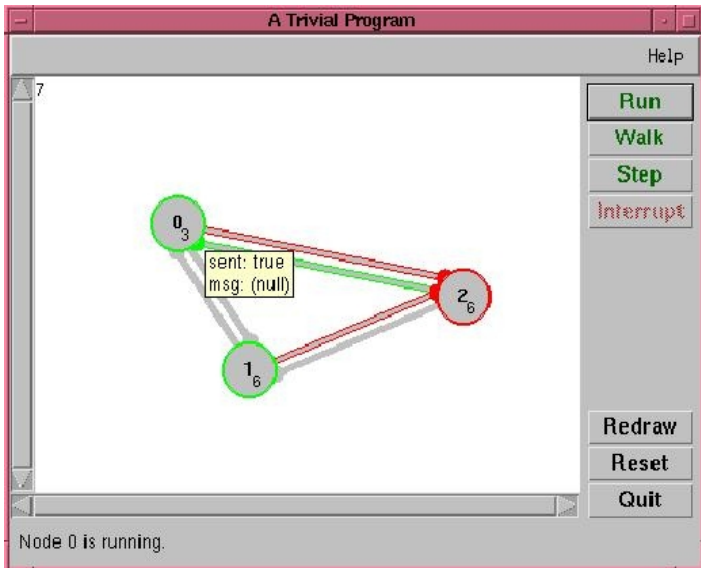

Fig. 22: Visualized execution in T-DAJ [Schr02]

DisASTer (Distributed Algorithms Simulation Terrain)

DisASTer (Distributed Algorithms Simulation Terrain) is a Java based platform aimed at providing an environment to simulate and visualize distributed algorithms, primarily in an educational context ([Got03],[OG05]). Processes in the system model are called *Agents* and are derived from a common base class. The system is driven through messages and follows a state machine design, where a global message queue acts in an similar manner to event queues found in discrete event simulators and event-based composition frameworks. The global message queue manages an individual queue for each agent in the system and allows for user manipulation of message ordering or removal of messages. Agents can receive different types of messages by overloading a base method *onReception(Message m)* with the corresponding message type. DisASTer uses Java *Reflection* to correctly identify the corresponding method for a particular message, as the *type erasure* found in JAVA would have removed the required information at runtime when messages are added to the (generic) global message queue. DisASTer is geared towards assisting students in the implementation and simulation of distributed protocols and algorithms covered in classwork. It allows for a coding style that more closely follows the algorithm descriptions found in textbooks than, for instance T-DAJ. The visualization component in DisASTer contains several different views available to the user. The *topology view* shows the underlying network structure with agents as vertices and connections as edges. It can display messages as they are communicated along links. The *sequence view* provides a sequence diagram of all messages sent by each agent. The *messagequeue view* shows messages yet to be delivered to agents. It is possible to define custom views and to modify or extend the aforementioned view elements. DisASTer provides a library of already implemented distributed algorithms relating to problems such as *Byzantine Generals* ([LSP82]) or *leader election*.

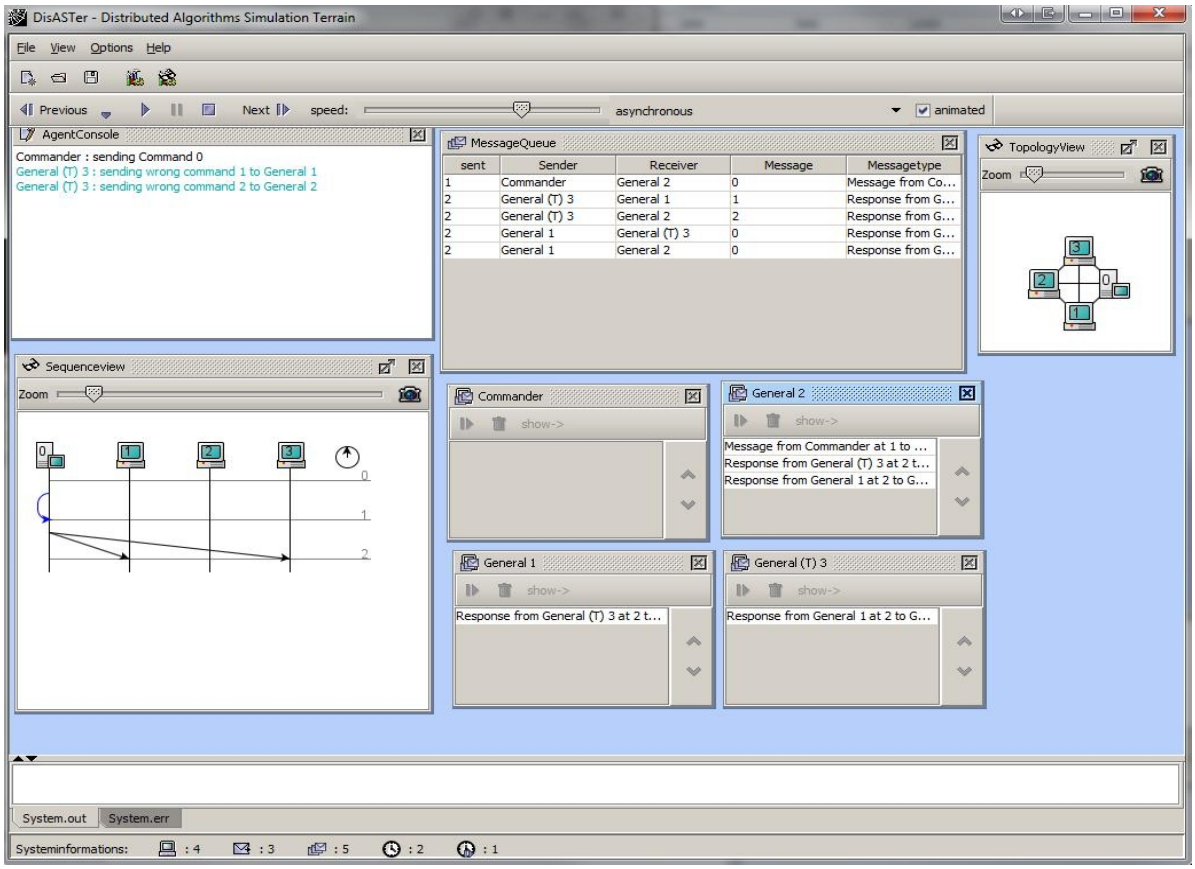

Fig. 23: Screenshot of a DisASTer Byzantine Generals example.

FADA (Framework Animations of Distributed Algorithms)

In ([O'Don06]) O'Donnell presents a simulation framework for teaching distributed algorithms that is described as:

"...informed by the strengths of algorithm simulation systems and algorithm construction systems, this thesis presents a simulation framework named FADA for the teaching and learning of distributed algorithms. The engagement model of FADA is that of a highly interactive simulation. Its underlying pedagogy is based on theories of constructionism and social constructivism. The design is also informed by theories of dual coding and epistemic fidelity." ([O'Don06]).

FADA's focus lies on enabling the development of interactive visualized simulations of distributed algorithms. It is Java-based and provides a development environment that includes a wizard to assist in the creation of new algorithms, a message passing algorithm library and an editor and debugger for modifying code. FADA can be used as an interactive presentation tool with already predefined algorithms or assist in the creation of own distributed algorithm simulations. The design relies on providing basic abstractions and visualizations for message passing distributed algorithms that can be extended to form a concrete implementation of a simulation for a particular algorithm. During the simulation's execution users can change variables or cause components such as messages or nodes to fail. It is also possible to control the speed of the execution. The simulation environment is based on a message passing model and mentions the use of a message queue, however the work ([O'Don06])

does not present any in-depth descriptions of the simulator architecture.[36] A *Node* class serves as the basis for modelling processes and implements the *Runnable* Interface. At the start of the simulation each node is initialized with its own Thread. From the provided code-examples it appears that the threaded execution of nodes is generally only used for initialization and initial message sending, whereas message reception and its processing is driven by the Thread invoking the *receiveMessage* method of nodes. The design of FADA is clearly aimed towards the development of educationally effective interactive algorithm visualizations rather than providing accurate or realistic simulations of distributed systems. This is reflected by its extensive coverage of educational theories relevant to algorithm visualization. In contrast to HiSAP, FADA also aims at simplifying the construction of new interactive simulations so that learners may themselves implement distributed algorithms rather than having to rely on an expert for their implementation. An exploratory case study was conducted to test the pedagogical and operational effectiveness of FADA. The results suggested that learners are motivated and engage with the system while instructors find it an effective and intuitive way to demonstrate and convey dynamic and concurrent behaviour ([O'Don06]).

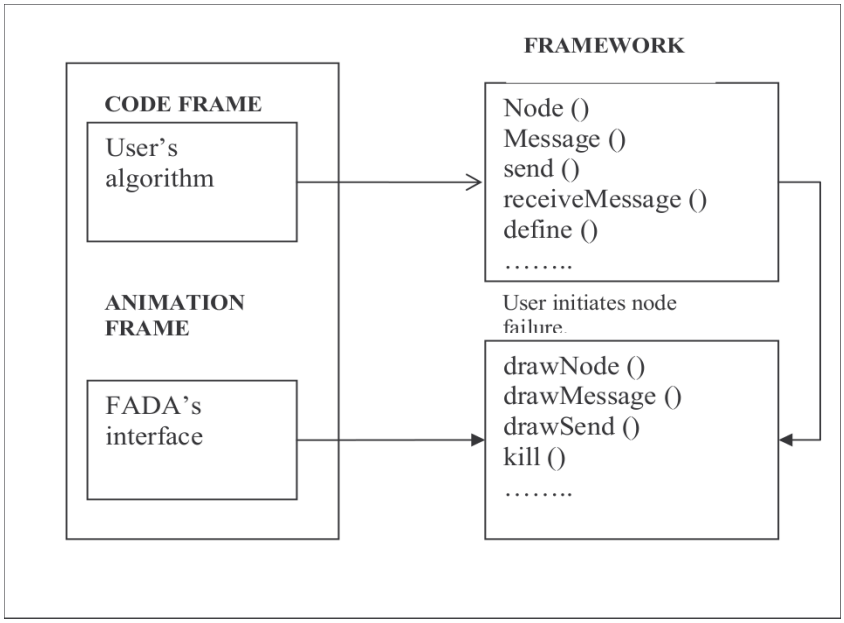

Fig. 24: Overview of FADA's modular architecture design. [O'Don06]

[36]At the time of writing the Author was unable to locate the source code of FADA for a better insight into its architecture.

2.5.3 Educational Framework Discussion

To gain a deeper insight into the requirements for developing an educational group communication system it is necessary to compare and discuss previous works addressing these issues. As already mentioned the topic area of group communication is rather specific and the author is unaware of works focussed on that particular aspect of distributed systems. Therefore the previous section (2.5.2) presented educational frameworks and simulation tools that cover more basic topics and concepts of distributed algorithms and protocols. While the complexity and requirements for the simulation of an entire protocol stack or GCS are generally more demanding than simulating a single distributed algorithm, the requirements on the governing pedagogy of both simulation environments should be similar. The rationale behind this assumption is that usually the general topic area of distributed systems is taught to more advanced computer science students, reducing the differences to those learners focussed on the particular topic of group communication.

An interesting counter-argument to the above stance can be made through the *expertise reversal effect*. The *expertise reversal effect* is a phenomenon that is explained with *cognitive load theory* ([Swe88]) and is backed by empirical evidence. In rough terms cognitive load theory assumes that individuals have a limited working memory where information is processed in the context of existing schemata that are stored in more long term memory. An excessive amount of new information may overload this working memory and have a detrimental learning effect.

“Because of the limited capacity working memory, the proper allocation of available cognitive resources is essential to learning. If a learner has to expend limited resources on activities not directly related to schema construction and automation, learning may be inhibited.” ([KACS03])

The expertise reversal effect suggests that additional information which may be helpful to novice learners for new schema construction can actually be counter productive for a more experienced audience. The additional information still needs to be processed in the working memory of experts even if it is not required, increasing the cognitive load without being beneficial for learning. A conclusion that can be implied from the expertise reversal effect is that learning materials should take the expertise of its target audience into consideration.

“The most important instructional implication of this effect is that, to be efficient, instructional design should be tailored to the level of experience of intended learners. Without such tailoring, the effectiveness of instructional designs is likely to be random. Indeed, recommendations to use particular designs can be potentially quite counterproductive.” ([KACS03])

Combining the findings on the expertise reversal effect with constructivist teaching methods is a highly controversial topic as cognitive load theory (CLT) and constructivism are often presented as conflicting epistemologies ([KSC06]). Discussions in this context may very well be influenced by the long standing differentiation between schools of thought. It is pointed out in ([Val02],[HB14]) that findings from CLT can also be relevant to constructivist learning theories and are not implicitly incompatible to each other.

The following brief descriptions of the in section (2.5.2) covered educational simulation tools outline their general educational approaches.

Lydian allows learners to define network topologies, starting conditions and failures. Own distributed algorithms can also be implemented through a custom language with c-like syntax, where different states and permissible state transitions need to be described for the algorithm. Parts of the visualization are handled by the tool, however algorithm specific visualizations are generally created by educators to capture the essential concepts.

HiSAP's architecture is designed to generate its simulation model from a provided formal specification, in this case using a variant of petri nets. The fundamental idea is to automate the generation of the (animated) visualization and simulation by having educators or learners provide a formal specification of the algorithm together with a set of correctness criteria ([PB01]). From this the system would either generate a simulation model with all possible scenarios or outline why the algorithm can't fulfil the criteria. In the presented prototypical system visualizations still have to be created manually and the formal specification has yet to accept provided correctness criteria. Simulations are stepped through where either a script defines all the algorithm's steps or the user interactively chooses from a set of valid options.

DAJ's focus lies on interactivity and a “full data display”, where the relevant data of each node of the system is visualized and students can choose from a set of predefined valid actions for each step of the simulation. A base class is provided with relevant GUI code to simplify the creation of new algorithms. This process is described as:

“The challenge in implementing an algorithm is to design and implement a state machine: in each state, the button panels and the processing to be done upon receipt of a message will be different. By using an existing program as a model, a new algorithm can be implemented in a day or two.” ([Ben-A01])

While students have also implemented new algorithms using DAJ this process appears to be directed towards educators rather than learners. It is important to note that Ben-Ari does not focus on elaborate visualization and animation of the algorithm's state transitions and is, in this regard, backed by the findings of Hundhausen et al. ([HDS01]) that suggest learner interaction rather than visualization fidelity is a key component for learning effectiveness. The mixed findings on the effectiveness of algorithm visualization have been discussed at the beginning of this chapter (2.5.1 Overview).

The design of **FADA** is informed by an extensive review and discussion of different learning theories in the context of computer science education and algorithm visualization in particular. At its core is the provision of an environment in which learners can interact with but also construct visualized interactive simulations of distributed algorithms. FADA places a bit more emphasis on the self-construction of distributed algorithms and their visualizations by learners, however the model still follows that of the other presented works where an expert is mainly responsible for implementations.

All of the above mentioned works have in common that their focus lies on conveying the abstract functionality of distributed algorithms and to aid learners in their understanding. By being able to observe and manipulate different states of the algorithm at multiple nodes students can refine and compare their mental model of the distributed algorithm with that of the implementation. The concrete implementation details or performance metrics are of secondary importance.

In contrast to these frameworks **T-DAJ** strives to provide an easy to use environment for students to implement distributed algorithms as a supporting exercise to the regular classwork. Visualization is generated automatically for displaying the general states of nodes and channels and can also return user specified text that describes the internal state. This visualization is primarily intended to better illustrate the global system state rather than to help convey the functionality of the algorithm with high fidelity AVs created by an expert. The programming model employed in T-DAJ resembles that of many real-world scenarios albeit being very simplified to reduce the learning overhead.

Similar to T-DAJ, **DisASTer** also offers students a simulation environment with reduced complexity where they can implement and experiment with distributed algorithms and protocols covered in the coursework. Its visualization component is more complex than that of T-DAJ but conceptually similar in that it is primarily used to display the global system state and events. Rather than following a process-oriented design DisASTer uses messages to drive computations. The interaction scheme is akin to event-driven systems where for each event a corresponding handler is executed. The pseudocode examples of distributed algorithms are often described using such event or message based interaction schemes making it more easy for students to translate a given algorithm to a working implementation for the DisASTer framework.

From the reviewed works two general approaches can be described. The first set of tools aims at teaching concepts of distributed algorithms and protocols through interactive visualizations, possibly in a distance education scenario. Here the focus lies on an accurate representation and communication of the underlying theoretical model to learners. It is assumed that the learner has either limited or no previous understanding of the distributed algorithm. The mechanisms of the framework are therefore geared towards illustrating the functionality of the algorithm by providing the user with an interactive, state-machine like model where the events leading to state changes (such as messages or failures) can be modified and the global system state is visualized to effectively convey this functionality. These simulation environments employ abstract system models to reduce the complexity of the algorithms' implementations. Furthermore, using abstract representations may avoid having to deal with platform specific issues or failures that are not directly related to the functionality of the to-be simulated algorithms. While some of the educational simulation environments also aid learners in the development of their own distributed algorithms rather than relying on implementations provided by an expert, the resulting code is tied to the environment and its goals of conveying the functionality. Lydian, DAJ, HiSAP and FADA can be placed into this first category of educational frameworks.

The second category of works places more emphasis on the implementation of distributed algorithms and the environments in which they execute. For learners at least a rudimentary understanding of the algorithm's functionality is usually assumed. Visualization is primarily used to display the global state of the simulation and provide more condensed and humanly better-interpretable information than textual representations such as log files. While these environments may also be used for teaching the fundamental concepts of an algorithm, they are more suited for experimenting with a concrete design or implementation. Such simulation environments hence are often used in the context of exercises or practical coursework where learners already have acquired a basic understanding of the algorithms. Their purpose is to ease the burden of setting up a distributed

system environment and reduce the learning overhead incurred when using complex real-world tools. From the covered educational works T-DAJ and DisASTer fall into this second category.

An educational group communication system would also fall into this second category of works as it is aimed at the implementation of multicast protocols and other, more complex algorithms and components rather than conveying general concepts through algorithm visualization and interactive examples. Here the focus lies on ease of use and data visualization to simplify the development and debugging process. A specific distinction between works of the second category and simulation environments targeted at experts and the scientific community is not made. It could even be argued that a neglection in the areas of usability of simulation environments actually leads to the necessity for dedicated educational simulation tools.

Mention of the expertise reversal effect serves to highlight that direct translations from one learner group to another may not necessarily yield the expected learning outcome for a particular tool or instruction method. This is not to say that the covered works and their employed approaches cannot be drawn upon for the design of an *e*GCS. It rather points out that further investigation in this direction is needed, both through surveys on the effectiveness of particular tool designs for different learner audiences as well as theoretical models that can better predict learning outcomes. From the initial argument it is not clear if learner expertise significantly differs between those that learn about general distributed systems topics and learners focussed on group communication concepts in particular. Furthermore it will strongly depend on the specific tool design if additional information is in fact counter productive to more experienced learners. A simple option to hide or display different forms of information within the *e*GCS tool may completely mitigate any potential ill-effects encountered through the expertise reversal effect.

2.6 Developing an Educational GCS: A Discussion

The initial goal of this work was the extension of a laboratory assignment and its simulation environment for group communication protocols to allow student developed multicast protocols to be used in realistic scenarios. These protocols also have to be viewed in the larger context of the enclosing GCS if students are to be able to rely on them for other parts of the assignment. For instance, the static group semantics of the current group communication simulator of the ADS laboratory (see Development of a Prototype Educational Group Communication System for a detailed description) will lead to implementations that are not trivially adapted to run in a dynamic group model. A GCS protocol stack and its composition model serves as a framework for protocol implementations and its design directly influences possible simulation approaches. Therefore the focus of attention is shifted towards GCSs and their potential usage in an educational scenario.

So far this thesis has covered both the fundamental concepts around group communication as well as literature that may inform the design of an educational tool for developing multicast protocols and other components of a GCS. Modern group communication middleware has moved from the monolithic constructs of early works to more modular and extensible designs. The management of components and individual protocol layers and their interactions are often governed by generic (protocol) composition frameworks, however ad-hoc solutions specific to a GCS are also commonplace. Either approach requires developers wishing to add new elements to have a thorough

understanding of both the composition framework and the individual components that make up the GCS.

These circumstances greatly complicate the application of GCSs in an educational context where learners only have a limited time to familiarize themselves with the framework and their understanding of the underlying theoretical concepts may be incomplete. Tools specifically designed to be used in an educational scenario that address these issues are mostly targeted towards less advanced topics. They either assist in the learning of the fundamental concepts of a distributed algorithm through letting the user interactively explore and observe its different permissible states, or provide a simulation environment where learners can easily implement such distributed algorithms. Both methods generally employ abstract system models to simplify the design and reduce the incurred learning overhead. To the knowledge of the author no educational tool[37] has been developed that explicitly covers the topic of group communication.

An educational GCS can be understood as a modular and extensible group communication system combined with an execution environment[38] that allows students to implement, test, and experiment with group communication middleware and its components. This discussion is centred around the question of how and even if an *e*GCS should differ from "regular" GCSs. Non-educational works, often targeted at the research community, exist that address some of the issues encountered in an educational setting. Their potential application in an educational context is often mentioned without providing specific details towards course design or what considerations need to be made if the software is to be used for teaching. Generally, the simulation of distributed middleware and modular protocol composition and group communication are treated separately, though some works such as Neko ([USD01]) or Kompics ([AH08]) may combine both aspects.

The execution environment needs to be considered as an important component of an *e*GCS. Distributed algorithms, and in this case distributed group communication middleware, is specified to be executed in a distributed environment. One way to offer such an environment locally is to simulate or emulate multiple processes and their communication channels on locally available hardware. Simulating a distributed system can allow students to experiment with different deployment scenarios and mitigates the potential problems of having to provide a real distributed system. The required level of realism afforded by the execution environment is up for discussion and will depend on the educational goals and usage scenario of the *e*GCS. It can range from an abstract simulation model such as those found in many of the reviewed educational tools to a detailed simulation or emulation of the lower level networking libraries of a programming language or operating system, which is often used for research, debugging and performance analysis. Depending on the particular design of the (*e*)GCS, executing multiple instances of the software locally may already behave very similarly to distributed counterparts. In such a case it might be sufficient to allow for fault injection for both channels and processes and possibly introduce some randomness to the system to achieve the desired level of realism comparable to a distributed execution.

The protocol composition framework or modular GCS is the other key element of an educational GCS as it dictates how protocol modules and components interact. In chapter (2.3) different composition schemes and protocol composition frameworks were presented. They are explicitly

[37]Primarily designed for an educational context rather than a tool highlighting its potential use in a teaching scenario.
[38]Possibly more than one execution environment may be used where some are dedicated to simulating and testing the GCS.

designed to simplify the development of complex protocols and protocol stacks and may even offer features such as an integrated simulation environment (Kompics, Neko) or stack composition through a GUI (SAMOA, see [Rüt09]). It is interesting to note that protocol composition frameworks are often presented in the context of developing a GCS. For one this may be explained by the complex interaction patterns between different protocols of the GCS stack that ask for a systematic approach towards designing a composition. ([PSM+07]) point out that professional IDEs are potentially too complex for introductory courses on programming and a similar situation can be found here in regards to (protocol) composition frameworks and modular GCSs. The complexity of these tools may require students to spend excessive time learning their functionality and hence render them impractical for many scenarios. It will need to be weighed carefully how much of the composition process is left to learners and how much of the core design of the *e*GCS is already provided, so that students merely have to extend or implement very specific components. A skeleton implementation may provide the necessary basis for student implementations without requiring them to have a full understanding of the entire model at the cost of limiting the possible routes of development a leaner may undertake.

The following discussion on developing an educational group communication system is presented as a set of questions where the answers are informed by the literature presented in this work. The first question deals with the educational requirements:

2.6.1 What are the design considerations for educational tool development for (advanced) distributed systems topics?

The review of literature on educational tools has shown a distinct lack of works for advanced distributed systems topics. Approaches towards designs for coursework in this area were covered by Wein et al. and Mishra ([WKC+09],[Mish02]) but neither focus closely on educational tool development. In an attempt to find suitable guidelines the scope was widened to include introductory courses on distributed systems and also simulation tools for distributed algorithms targeted at these entry level courses. Unfortunately even with such a broadened search space there are still few publications on how to structure course design and develop such simulation tools. In (PSE+05]) Pears et al. point out that there is no commonly recognized core literature for *computing education research* (CER). This lack is not just an issue for researchers working in the field, but also for educators trying to base their tools and coursework on such research. Approaches for structuring courses and designing educational tools outside of computing education are not trivially translated to computer science topics. For instance the field of algorithm visualization (AV) tries to address many of the inadequacies when using traditional methods for teaching and illustrating the behaviour of algorithms such as using “chalk and talk” presentations on blackboards ([O'Donn06]). However findings on AV effectiveness are mixed, suggesting learner interaction as key factor for creating educationally effective visualizations ([HDS01],[NFmN+02]). Without knowledge of these findings educators may use or design ineffective or inadequate tools.

It is beyond the scope of this thesis to attempt to define a core literature for informing the development of educational tools for (advanced) distributed systems topics. However as a first step in this direction and to identify possible guidelines for developing an educational GCS an extensive set of educational (simulation) tools for distributed algorithms was covered. Although these works

are aimed at a less experienced audience many of the problems faced also apply to developing an educational GCS. From this informal survey two design aspects are pointed out that could be found in most educational tools, namely an *interactive learning environment* that provides feedback to learners and the use of *visualization* to better illustrate algorithmic behaviour. Furthermore works on course design often used *realistic problem scenarios* to engage students in activities similarly to those they would encounter in “real-world” situations.

Provision of an Interactive Learning Environment

Giving students the ability to dynamically interact with and explore the behaviour of an algorithm is an approach supported by constructivist learning theories. Through this interaction learners can refine their internal model of the algorithm and are encouraged to discard incorrect views once the exhibited behaviour is in dissonance with their projected expectations. Generally, but not exclusively, the provision of interactivity in the reviewed tools was argued in the context of visualized simulations for distributed algorithms. It makes sense to discuss interactivity in this manner as the visualization component can provide an overview of the relevant states of all components of the system. The effects of changing parameters or otherwise modifying the execution behaviour may be more readily noticed through visualization. Interactivity in the context of these learning tools is mostly tied to how flexible the simulation model is in combination with the specified distributed algorithm as well as the provided options to manipulate the simulation state. Realism in the simulation model is of secondary importance as long as the essential concept and properties of the algorithm are adequately conveyed.

It is not implied by interactivity that learners need to interact with a fully functional or even correct simulated algorithm. If we are to follow constructivist reasoning learners can also benefit from other types of engagement that demonstrates behaviour contrary to their expectations. Fekete ([Fek03]) for instance describes the use of *counter-examples* in coursework, in this case for the topic area of data structures. These counter-examples are built around common misconceptions and students are to actively engage with the problem by identifying and correcting programming and design mistakes of provided code as well as giving an explanation of why that mistake was made. Such a learning approach engages students in higher order thinking and *meta-cognition* as well as requiring them to refine and construct a new internal model of the program or algorithm if it has deviated from the expected behaviour. When using counter-examples students are actually faced with incorrect learning material, but are given the resources and freedom to engage with the problem and find a solution. The interesting aspect in this case is that counter-examples and the debugging of code are conceptually very similar. If students can be made aware of incorrect behaviour of their own code or are provided with other student's implementations and have to show if they are correct or not, it stands to reason that this approach can be similarly educationally beneficial to them.

The type of interactive learning environment provided in an educational GCS is intimately tied to the environment in which it executes. An *e*GCS and its contained components should exhibit similar behaviour to that found in real systems and also provide easy methods for specifying external events such as process crashes or link failures. Students should also receive useful debugging feedback to assist them in their learning process. Simulation or network emulation is an obvious approach towards allowing the local simulation of a distributed application such as a GCS. One difficult aspect to ensure is the deterministic behaviour of the system throughout consecutive executions

when using the same parameters. As GCS protocol stacks are complex and possibly execute concurrently it may become impossible to guarantee a deterministic execution for the exact same set of starting parameters ([FKK+05]). In this regard a more abstract simulated environment may provide the advantage of completely deterministic behaviour at the cost of hiding potential concurrency issues and other effects that may arise during “regular” execution.

An interactive learning environment in the constructivist sense should be perceived as one where learners can experience feedback from their actions and this feedback is realistic and akin to what would be observed in “real” situations. A certain degree of freedom is required so that learners can fully explore and contrast their mental model to that of the implementation in their own terms, rather than relying on an expert to have already made the essential associations. In an educational GCS learners hence need to be made aware of incorrect behaviour and, more importantly, the simulation environment needs to be accurate enough to return realistic and useful feedback. A model that is too abstract or allows very little external modification may not be sufficient to highlight incorrect internal representations.

Visualization

Algorithm Visualization (AV) has long been used as a tool to aid learners in their understanding of algorithmic behaviour and is a central element in most of the reviewed works. However, as was pointed out in the section on educational simulation tools (2.5.2), the educational effectiveness of AV has been the topic of ongoing discussion. Of particular interest are the findings by Hundhausen et al. ([HDS01]) that suggest learner interaction plays a key role in the effectiveness of algorithm visualization. Augmenting traditional teaching methods by letting learners passively view (pre-built) visualizations is not expected to provide any significant learning advantage over using no visualization at all. Many of the presented tools assume that visualizations for a particular distributed algorithm or problem are partially or fully constructed by the educator. Some provide libraries where certain visualization components can be re-used and might even be drawn upon by students to construct their own visualizations ([O'Donn06], [KPT03]). In these models the simulation environment and visualization are closely linked and rather specific to the problem domain of the distributed algorithm.

Visualization can also be used to render available data structures and information more human-readable. For instance Ben-Ari ([Ben-A01]) uses the visualization component as a means for giving “full data display” and not as a high fidelity abstraction for illustrating concepts. Simulation tools not explicitly designed for educational purposes such as DAP ([CKP+04]) may also employ visualization as a means of displaying information and allow users to interactively change the simulation state. In this regard the goals for both educational and non-educational scenarios are similar where visualization is used to more effectively illustrate important information rather than to convey an expert's mental model.

“Since distributed algorithms are complex abstract objects, visualization plays an important role in improving their perception and their understanding. Researchers in distributed computing need visualization during the design and debugging phase of an algorithm and also when presenting the algorithm.” ([GMB00])

A group communication protocol stack is not trivially visualized in a meaningful way, and different components of the stack such as Consensus and multicast protocols might require different forms of visualization if they are to aid in the understanding of the underlying concepts. Basic forms of visualization such as those used in ([Schr02],[CKP+04],[BWWZ05]) can highlight the state of nodes and communication links and provide some feedback for debugging, especially if users are able to pass important information on to the GUI (i.e. providing a textual representation that is displayed for each node).

Realistic Problem Scenarios

The third design consideration for an educational GCS stems from the literature on designing distributed systems coursework in general. Rather than giving students abstract exercises that do not reflect the reality of their later professional careers an educational tool should try and offer realistic problem scenarios. What this means is that these exercises should be formulated as a larger problem that students can explore and try to solve through their own research. The instructional method of *problem based learning* (PBL) offers guidelines on how such coursework might be structured. In regards to an educational tool for implementing group communication components the difficulty arises on how to give students ample freedom to explore a specific problem while still being able to assist them in the creation of functional group communication protocol stacks. One possibility is to provide a basic skeleton composition of a GCS where students can choose for which area they want to make specific improvements or modifications. The other is to give students a more complex task as an assignment where the only requirement is to base the design on the provided *e*GCS the solution. If the group communication system only contains minimal functionality students are forced to implement their own improvements to satisfactorily solve the larger problem they are tasked with.

The second question asked addresses the requirements for a tool that satisfies the set out problem.

2.6.2 What are the requirements for an educational tool that supports the development of group communication components and multicast protocols that can be used in a realistic scenario?

Relevant aspects of developing such an educational tool have been covered in this work in separate topic areas. In particular, group communication systems and protocol composition, the testing and simulation of GCSs and educational simulation environments for distributed systems topics were addressed. If students are to be able to use their developed protocols in realistic scenarios, the educational tool should provide the following list of features. The specified requirements are intentionally held very general to allow for a broader range of possible scenarios.

- The tool should support dynamic group semantics. Dynamic groups expose learners to essential aspects of group communication such as the group membership problem and virtual synchrony.
- The composition model needs to be modular enough to allow for an easy integration of student protocols into the GCS protocol stack. The development of additional multicast protocols should not require students to have an in-depth knowledge of the entire system and follow easy and straightforward guidelines.

- Provided components should have simple, well defined APIs. Students will have to rely on different components of the GCS such as failure detectors or more basic message passing protocols when developing more advanced protocols. The functionality of these building blocks needs to be clear and straightforward.
- Multiple instances of a protocol stack should be easily executed on a single machine at the same time, simulating a real distributed system. It cannot be expected of students to maintain their own distributed systems testbed for testing and debugging purposes. Providing students with solutions such as Emulab ([HRSD+]) or PlanetLab ([CCR+03]) or running a dedicated distributed system can furthermore be time consuming and costly.
- The communication and failure models of the local (simulated) execution should be realistic enough to ensure that protocols exhibit the same, or at least a very close range of behaviour as encountered in a real environment.

As a third question the necessity for a specific educational tool is considered:

2.6.3 Can the requirements for an educational GCS be satisfied through existing non-educational works?

Most of the surveyed educational tools are aimed at introductory distributed systems courses. It is assumed that their target audience has little or no experience within the topic area. On the other hand, the students that are in the focus of this work are expected to have a solid background in software engineering and at least a basic understanding of distributed systems concepts. In this environment the distinction between learner and researcher becomes blurred as students cover state-of-the-art topics and develop their own distributed algorithms. Learning theories and their guidelines are of course not limited to less experienced students and also apply to the target audience of this thesis. A fundamental issue encountered with most protocol composition frameworks and modular GCSs as well as the tools used for their simulation lies in their design. They are geared towards the research community or commercial applications rather than aiming for easy to understand concepts and structures. What this means is that these frameworks offer advanced functionality and extensibility at the cost of being complex and requiring a significant learning overhead. Burri et al. for instance highlight this issue in regards to teaching and network simulators:

“Above all when used for rapid prototyping or for educational purposes—such as in exercises accompanying a lecture—the complexity of network simulators can turn out to be a major drawback. The effort and time invested in understanding a full-grown simulator often stands in no reasonable relation to the yielded profit.” ([BWWZ05]).

Many of the reviewed works mention their potential application in an educational setting but the problem of learning overhead is usually not addressed. Some works, such as Kompics ([AH08]), place a stronger focus on the educational aspect while others (Neko [USD01], DAP [CKP+04]) use designs that appear more readily adaptable to follow a design similar to the presented educational tools in chapter (2.5).

From the previous question learner interactivity, visualization, and realistic problem scenarios were established as key aspects for an educational tool in this topic area. Of the reviewed non-educational frameworks and tools no single work addresses all of these aspects while also providing all

necessary components for a fully functional GCS. The combination of several different tools is of course possible but further increases complexity and learning overhead. In an answer to the posed question it appears that at least some modification of existing non-educational tools is required to provide the desired functionality of an educational GCS.

2.7 Summary of Part One

The general problem statement of this work is the development of a tool for teaching advanced distributed systems concepts. Clearly any design should be based on research in both the topic area of interest, in this case group communication, and works focusing on educational requirements for such tools. This thesis was neither set out to build a core literature for the design of educational tools for advanced distributed systems topics nor was its intention to perform a meta-analysis of works to gauge educational effectiveness such as the survey on algorithm visualization conducted by Hundhausen et al. ([HDS01]). Unfortunately the lack of such a core literature meant that at least some form of informal overview was required if the chosen design approach is to be informed by previous findings. One of the biggest difficulties lies in estimating the educational effectiveness of different tool and course designs. Available educational works in the topic area of distributed systems are primarily geared towards entry level courses. Few offer accompanying learner surveys to estimate the educational impact of the tool beyond anecdotal evidence. Furthermore a fundamental question that needs to be asked is if the results and experiences of such available educational tools can be directly transferred and applied to more advanced topic areas such as group communication. To what degree or even if the expertise reversal effect discussed in section (2.5.3) can be expected to apply to the approaches of the covered educational frameworks is unclear without further investigation. Giving learners the ability to add or remove additional information or other elements intended to aid in the learning process in an educational tool might mitigate most of the described effects.

The above mentioned problems leave many unanswered questions on the development of educational group communication systems, and educational tools for (advanced) distributed systems topics in general. Without further research on the effectiveness of a particular design approach a lot of the design decisions have to be based on reasoning rather than factual evidence.

This part of the thesis has provided the reader with an overview of the topic area of group communication, a closer look at the development and simulation of group communication systems through the lens of a researcher or developer, educational works that deal with the general topic area of distributed systems, and finally a discussion on how this information might be used to construct an educational tool for the topic of group communication. It is a first step towards identifying core literature for developing educational software tools for advanced distributed systems topics such as group communication.

Part Two

3. Development of a Prototype Educational Group Communication System

3.1 Overview

Part two of this thesis outlines the development of an educational Group Communication System (*e*GCS) prototype aimed at extending an existing group communication simulator for an advanced distributed systems laboratory assignment. In particular the prototype is to support dynamic group membership and virtual synchrony and allow students to use their self developed protocols in realistic scenarios. The design decisions are informed by the findings of part one (see Educational Software Tools for the Topic of Group Communication). Chapter (3.2) provides an overview of the current group communication simulator and the advanced distributed systems (ADS) lecture and discusses the general motivation of why the simulator should be extended. Two primary goals, namely the possibility to use student developed protocols in realistic scenarios with little or no modification and the extension of the simulator to support dynamic groups and virtual synchrony are defined. In (3.3 Findings of Part One applied to the set out Goals) the set out goals are discussed in light of the findings presented in part one. Visualization is currently not considered and will have to be addressed once the prototype is fully functional. The aspect of interactivity is covered through testing and debugging and follows a similar reasoning to the *counter-examples* used in ([Fek03]). As for the question of realistic problem scenarios an approach using *problem based learning* may be too extensive for the limited time frame available to students for completing the laboratory assignment. Chapter (3.4) offers a general design proposal where, rather than extending an existing GCS, the decision is made to create a new *e*GCS prototype based on the header-driven model for protocol composition and Consensus as a basic building block for primary component group membership and view synchronous multicast communication. Different methods for simulation and testing are also discussed. A similar approach to that described by Farchi et al. in ([FKK+05]) is chosen for testing while simulated executions are to rely on the distributed middleware simulation framework MINHA ([CBCP11]). Finally chapter (3.5) outlines the development of the *e*GCS prototype and highlights implementation details as well as issues that were encountered using the proposed design. In regards to simulated executions initial results using MINHA are promising, however it was concluded that the framework is not yet mature enough to be used in an educational context. Alternatives to allow local simulated executions are therefore pointed out. The development process also involved implementing the modality of the first part of the original ADS laboratory assignment (development of asynchronous static group reliable multicast protocols) using the header-driven composition model, parts of the eGCS stack, as well as the described testing approach. (3.6 Outcome) offers a brief overview of the outcome of part two of this thesis.

3.2 Extending an Educational Group Communication Simulator

3.2.1 The Advanced Distributed Systems Lecture

The motivation for this thesis is based on extending the laboratory assignment for an *advanced distributed systems* (ADS) course ([GF11]) held at the Vienna University of Technology. To gain a better understanding of the target audience and the specific problems and requirements for this particular assignment the course contents are briefly outlined. The lecture introduces more advanced distributed computing topics to computer science students participating in master's programs at the TU-Wien with a special focus on the conflicting priorities of flexibility and dependability ([GF11]). Students are expected to have acquired a basic understanding of distributed systems concepts beforehand. A rough overview of the ADS lecture is given on the following lecture slide from ([GF11]).

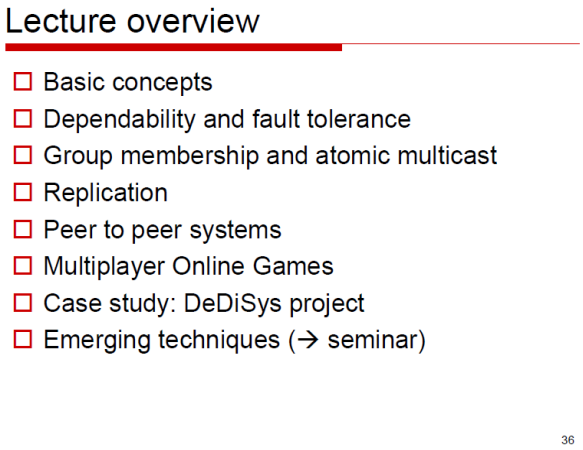

Fig. 25: Overview of the Advanced Distributed Systems Lecture

The course is structured into three parts. In the *lecture* theoretical concepts are explained but also derived through audience interaction and discussion. The *seminar* component requires participants to extend the discussed topics by presenting state-of-the-art methodologies and research on a chosen subject area. Here small groups of around four participants work together as a group and present such a topic. Finally, in the *laboratory assignment* key topics of the lecture are extended and intensified through an integrated project assignment. The goal is to provide hands-on experience with the implementation of concepts for the development of fault-tolerant distributed systems.

3.2.2 The ADS Laboratory Assignment

Currently the laboratory consists of two separate tasks. In the first part students are required to implement broadcast protocols with different guarantees within a provided group communication simulator. The simulator handles both the message passing functionality of the transport layer as well as simulating processes on the application layer as they communicate with each other using the

group communication middleware. Students are tasked to implement protocols with the following guarantees[39] or need to explain why they believe the particular guarantee cannot be implemented[40]:

- Non-uniform reliable broadcast
- Uniform reliable broadcast
- FIFO reliable broadcast
- Causal reliable broadcast
- Atomic broadcast

The simulation environment provides reliable asynchronous unordered point-to-point message communication between processes where processes can fail in the *crash-stop* model. Students can also choose to weaken this model by specifying a probability $p < 1.0$ for a channel to successfully deliver a message and hence simulate unreliable channels where messages can be lost. A basic unreliable multicast primitive is provided in the simulator's API where both sending and receiving processes can fail during a multicast[41]. The group membership component of the simulator assumes a *static group* model where all processes agree upon the initial set of processes that constitute the group. Students are required to implement their own unit tests alongside their protocol modules that check if the specified guarantees are actually met. For this a *TestProvider* Java class needs to be extended which defines initial conditions and events for simulation runs and returns a list of all processes (including those that failed) and the events they observed after the simulation has completed. Using this information students need to define *distributed invariants* akin to the approach described by Farchi et al. ([FKK+05] see section 2.4.2) that assert the correctness of the protocol and its specified guarantees and return a boolean value if the protocol is deemed correct or not by the TestProvider.

For the second part of the assignment a fault tolerant, distributed version of a Java-based implementation of the board game *Alcatraz* ([Web13]) is to be developed. The problem is split into two tasks, the server and the client implementation. For the *Alcatraz-Registry-Server* students need to ([FG11]):

- Design and implement a replicated registry service for a coordinated game initialization on the basis of the theoretical principles discussed in the lecture. The type of replication protocol is up to the students however synchronization and consistency needs to be guaranteed.
- Allow the registration and start of games as soon as at least one registry server is active. Players registered for a game shall be able to check out as long as the game has not commenced.

[39]The reader is directed towards section (2.2.5) for an overview of different multicast and broadcast guarantees.
[40]The communication model is asynchronous which would require students to implement their own failure detector abstraction in a mode of partial synchrony to be able to implement reliable total order (atomic) broadcast (uniform reliable broadcast can be implemented if we assume $f < \lceil n/2 \rceil$). The current simulator is not designed to support such timing assumptions.
[41]Multicast is implemented as a sequence of unicast operations and it is not failure atomic, i.e. a sender can fail halfway through the multicast so that only a subset of correct processes receive the multicast message.

- Failure of a registry server should not result in the loss or abortion of running game registrations. Such failures and the fact that replication is taking place is to be rendered transparent to the client. Single point of failures are to be avoided.
- The GCS *Spread* ([AS98]) is intended to be used for the synchronization of running servers.

The *Game-Client*'s requirements are specified as follows ([FG11]):

- Through user interaction the client has to perform a coordinated game initialization where a game consists of two to four players.
- Player names should be exchanged between game-clients and be unique for a particular game.
- Game moves are to be directly exchanged between game-clients without any server interaction.
- Short network outages are not to cause incorrect behaviour of the game such as crashes or deadlocks. Recovery of the connection or from a client crash should allow for the continuation of the game.
- Communication between client and registry server and respectively between clients is to be realized through Remote Method Invocation (RMI).

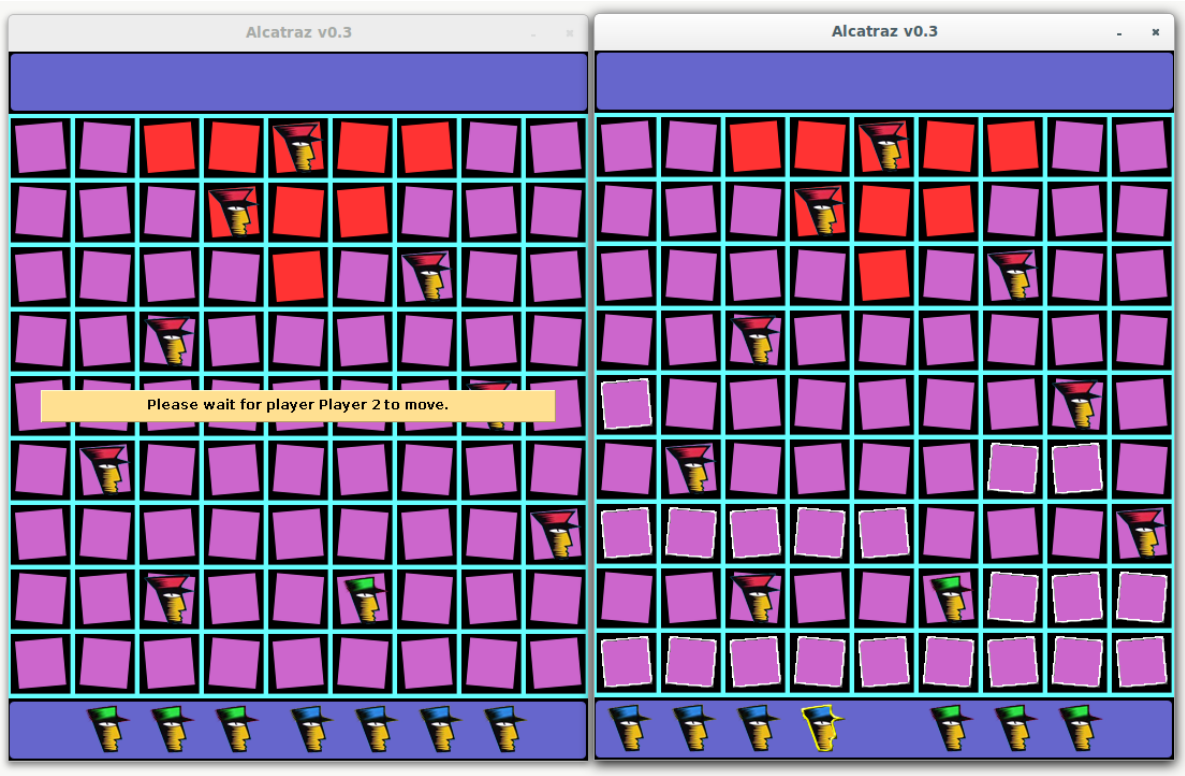

Fig. 26: Screenshot of the Alcatraz game being played by two players

3.2.3 Adding Dynamic Groups and Virtual Synchrony

The main motivation that prompted this work was the extension of the current simulation environment so that students can use the protocols they have developed in the first part for the second part of the assignment. While the provided simulator presents the learner with an abstract system model that can greatly simplify the creation of multicast protocols it also hides many important details. In particular dynamic group membership is a desirable feature that is currently available to students in the second part of the laboratory assignment through Spread. Spread is a partitionable group membership service that provides *extended virtual synchrony* ([MAMA94]) and utilizes a client-daemon model with an asynchronous FIFO buffering scheme[42] so that clients can engage in non-blocking virtually synchronous communication with a *same view delivery* guarantee.

In its current form the group communication simulator allows students to implement multicast protocols in a manner that is similar to many of the formal specifications provided by the scientific community which often use a static group model, as it factors out the added complexity of dealing with dynamic groups. Offering virtually synchronous group communication primitives can simplify the adaptation of (static group) multicast protocols for the dynamic group model, however at least some degree of modification is still required. For instance for message ordering guarantees it needs to be defined if they only apply to the messages within a view or across views (which implies some form of state-transfer or actualization mechanism for newly joined processes). In the partitionable model protocols also need to be able to deal with the merging of different partitions and consider that concurrent partitions have diverging states.

The added complexity incurred if dynamic groups and virtual synchrony are added in the first part of the laboratory is warranted by the additional flexibility it offers and also the educational merits of including and highlighting this important aspect of group communication. From the initial intention of allowing students to use their own multicast protocols in later parts of the assignment the following two goals for extending the current simulation environment arise:

- Possibility to use student developed protocols in realistic scenarios with little or no modification
- Extension of the simulator to support dynamic groups and virtual synchrony

The next chapter (3.3) will discuss how these two goals may be achieved in the context of the findings of part one of this thesis on developing educational tools for advanced distributed systems topics.

3.3 Findings of Part One applied to the set out Goals

Part one of this thesis extensively covers the topic of group communication and highlights many of the issues encountered when considering the development of educational tools for that particular area of interest. Therein it is pointed out that little directly related research on the design of such educational tools for the development of software components related to group communication exists. The in part one derived design considerations for an *educational group communication system* (*e*GCS) are based on general observations of approaches and methods found in the more

[42]The daemons themselves actually block and provide sending view guarantees but client requests are buffered while a view change takes place

general field of distributed computing. In particular the provision of an *interactive learning environment*, *visualization*, and *realistic problem scenarios* are highlighted as design decisions often encountered in educational works. Currently the ADS laboratory provides a realistic problem scenario through the second part of the assignment where a lot of the specific design decisions are left up to students. The first part of the assignment offers some degree of interactivity through repeated simulation and debugging of protocols and requires students to also think about failure scenarios which can, to some degree, be compared with the *counter-examples* given by Fekete et al. in ([Fek03]). Visualization in the current group communication simulator is not considered.

3.3.1 Educational Design Decisions for the ADS Laboratory

Visualization

In respect to the three commonly found educational design decisions mentioned before, namely an *interactive learning environment*, *visualization*, and *realistic problem scenarios* it is argued that visualization plays a secondary role during the prototype development. We have to discern between visualization that is aimed at teaching and intensifying the understanding of an algorithm or class of algorithms (*algorithm visualization)* and visualization that is primarily geared towards presenting data in a more human-readable form. In part one the overview of relevant educational tools (2.5.2 Educational Simulation Tools for Distributed Algorithms and Protocols) shows that works focused on teaching the fundamental concepts and functionality of an algorithm generally tend to use algorithm visualization (AV) while on the other hand tools that aim at providing a (simulation) environment for executing and testing algorithm implementations, a situation where learners generally have at least a basic understanding of the algorithm's functionality, often employ forms of data visualization. Here this differentiation is primarily made to outline that the creation of algorithm visualizations is usually not an automatic process but rather relies on an expert to correctly highlight important aspects and choose a suitable form of illustration. A data visualization tool may also be suited to help convey the functionality of an algorithm but in this case the tool will most likely not have been specifically designed for the particular algorithm of interest. In regards to the ADS laboratory where students should be able to implement a variety of algorithms and protocols the second method is more feasible. Furthermore it is up to students to chose a concrete algorithmic approach from the various solutions provided by the research community or even come up with their own design. This scenario renders the provision of algorithm visualization tailored for a particular implementation extremely difficult. A solution might be to task students with the creation of their own visualizations as they will have to have at least a basic understanding of the algorithms they are implementing. Such an approach for instance is also considered in ([O'Donn06]) and from a constructivist perspective the self-creation of AVs should be educationally beneficial to learners. Applying these observations on visualization to the proposed goals for extending the laboratory simulator, also in respect to the target audience, suggest that visualization efforts should be directed towards effectively illustrating data in order to assist the development and debugging process rather than providing AV to aid in the learning of an algorithms functionality. As indicated by Ben-Ari a “full data display” ([Ben-A01]) might be more important than elaborate algorithm visualization, also in light that the theoretical concepts are presented in the lecture beforehand. Part one also outlines findings that suggest the educational effectiveness of (algorithm) visualization may be dependent on the interactivity it provides to learners (see Algorithm Visualization in section 2.5.1). If this

interactivity cannot be ensured the efforts put into designing and integrating visualization may yield little to no advantages for learners. Without a fully functional educational tool that actually supports the development and testing of multicast protocols and other GCS components the educational benefits of including visualization are questionable. The integration of visualization, either to illustrate data or as algorithm visualization are therefore postponed until a working prototype of the extended group communication simulator exists, after which the topic needs to be revisited.

Interactivity

A form of interactivity is currently achieved through the development and debugging process of protocols in combination with the simulation environment. Like the *counter-examples* in ([Fek03]) this process requires higher order thinking and can can cause cognitive dissonance when the learner's mental model does not correspond to the perceived behaviour of the protocol or GCS component. Through this interaction cycle of debugging and modifying code and then observing how these changes affect the execution learners can refine and correct their internalized mental model. Simulation and testing capabilities are therefore an integral aspect of a learning environment and an important element of an *e*GCS. The envisioned prototype design should therefore make careful considerations to adequately and effectively include simulation and testing facilities.

Realistic Problem Scenarios

Offering realistic problem scenarios with a wide scope that enables learners to explore the problem domain and potential solutions with relative freedom is an instructional approach that can be argued through constructivist learning theories (see Cognitive Constructivism in section 2.5.1). For the current ADS laboratory this is primarily found in the second part where a fault-tolerant distributed game is to be implemented. The extension of the simulator to support dynamic groups and also take into account design considerations of an *e*GCS allows more freedom in the development of multicast protocols and other GCS components. More realism is added to the first part of the assignment as students now have to consider how their implementations may affect the later task and the entire assignment forms a more coherent unit. It may appear logical to extend the problem even further by having students develop their own GCS rather than just the multicast protocols that are to be used by the GCS for the second part, however the current approach of well defined smaller problems seems better suited for this particular laboratory. In part this has to do with the limited time frame in which students are to complete the assignment. Problem based learning requires that students have ample time to research about the given problem and come up with potential solutions. Tasks that are too broadly defined may set learners on a path where it becomes impossible to find a solution in the given time. Especially in regards to group communication and developing a GCS the learning overhead can be high and required development time may vastly exceed expectations. A more tightly defined exercise such as requiring the implementation of different types of multicast protocols satisfying various guarantees using a well defined API provides a more predictable outcome. Furthermore it makes it easier to supply students with reference implementations for particular components if they fail to solve the given problems on their own. This is especially important if later parts of the laboratory such as the registry-server of the game are changed to rely on students' earlier work rather than on the GCS *Spread*. Grading can also become more difficult if learners are given more freedom in their approaches.

3.3.2 Supporting Dynamic Group Membership and VSC

First we address the the goal of how dynamic group membership and view synchronous multicast communication (VSC) may be integrated into the laboratory and its group communication simulator. A straightforward solution would be to extend the simulation environment so it emulates view synchronous group membership and its guarantees like the approach presented by Drejhammar and Haridi ([DH12b]). Such a method allows for the extension of the current simulator design without requiring significant changes to the overall structure. The API would need to be modified so that protocols receive notifications of view changes and can query the simulated view synchronous group communication protocol layer on information such as message stability or the current membership set. Testing of protocol modules could be conducted in a similar manner to the current design that relies on TestProvider classes. The obvious downside to such an approach is that the provided VSC layer and group membership service are part of an abstract simulation and require fully functional counterparts if protocols should be usable in realistic, non-simulated scenarios. Another approach involves the development or adaptation of a fully functional (dynamic group) GCS to readily support the development and integration of additional multicast protocols and where testing is conducted using the same GCS. Either specific communication layers in the stack are replaced by simulated counterparts or the entire GCS stack is executed in a virtualized environment. One issue when choosing this method is the necessity to account for timing assumptions made by various components and layers in the stack. A fully simulated VSC layer like the previously described solution may either provide the failure detector mechanism through its membership updates or by also supplying a simulated failure detector abstraction[43], requiring no additional timing assumptions. If a regular GCS is used great care needs to be taken to account for any components that rely on time. This also adds additional complexity to the definition of simulation scenarios as they will, to some degree, also have to use explicit timing such as specifying how far the simulation should advance in simulation time. The reader is referred to part one where the problems involved in simulating and testing GCSs are discussed in more detail (2.4 Simulation and Testing of Group Communication Systems).

Another interesting question in regards to dynamic group membership arises when we consider the two different forms, namely *primary component group membership* (PCGM) and *partitionable group membership*. In an educational context the PCGM model may be better suited if students are to develop their own multicast protocols. While the partitionable model improves availability of the system its specification and the development of protocols that effectively use extended virtual synchrony can be more complex and time consuming. In ([Sch02]) it is outlined that PCGM can trivially be implemented on top of Consensus, a basic building block for many distributed protocols and GCSs. Considering the possibility of later extending student tasks to include the development of their own Consensus protocols, group membership service and basic view synchronous multicast communication protocols the primary component model may incur less learning overhead. Adaptations of static group algorithms are more straightforward if only a primary partition exists and agreement on messages seen in views is provided through virtual synchrony.

[43]It can prove to be advantageous to provide both a failure detector abstraction as well as group membership rather than rely on the membership views for failure detection. [Sch02], [ST06]

3.3.3 Supporting the usage of student developed protocols in realistic scenarios

This brings us to one of the primary goals for part two of this thesis, namely the ability to use student-developed multicast protocols not just in the simulator but for later parts of the laboratory assignment. If we ignore the additional goal of wanting to support dynamic group membership the current simulator can be readily extended to provide this feature. It requires the implementation of an initial bootstrapping mechanism so that the static group can be set up and started remotely as well as bidirectional quasi-reliable point-to-point channels between all members of the group that replace the simulated links during regular execution. As outlined by Chandra and Toueg in ([CT96]) algorithms can avoid having to use timing assumptions by moving the requirements for models of partial synchrony to the failure detector abstraction. To be able to implement total order broadcast, Consensus or other harder problems a distributed failure detector needs to be included for both the simulation environment and the regular execution mode.

For the dynamic group model the complexity is increased as student developed protocols rely on the fundamental abstractions provided by the GCS, namely a group membership service and virtually synchronous communication primitives. As outlined in the previous section there are different possible routes for the provision of a simulation environment supporting dynamic groups and view synchrony. However either approach requires a fully functional GCS once protocols are to be used in real scenarios. In part one several GCSs and modular protocol composition frameworks are covered and their general designs highlighted. Currently there exists no GCS that completely satisfies the requirements set forth for an *e*GCS, so at least some degree of modification or adaptation is needed. Ideally an *e*GCS would combine the ease of use, visualization, and interactive simulation aspects of the in part one presented educational tools with the functionality and extensibility of a modular GCS based on a generalized (protocol) composition framework. The simulation model plays an important part in this design decision as both the GCS and additional educational components have to rely on it. A first step for the design proposal would hence be to look at different available simulation tools or protocol composition frameworks that include a simulation environment and use them as a basis.

After having outlined how the findings of part one of this thesis apply to the set out goals for extending the current ADS laboratory and its group communication simulator the next chapter offers a design proposal for the new prototype.

3.4 Design Proposal

3.4.1 Overview

This chapter describes the proposed design of an *e*GCS prototype for the in section 3.2.2 previously outlined advanced distributed systems laboratory (3.2.2 The ADS Laboratory Assignment). The composition of protocol stacks is based on the header-driven model ([BMN05],[Men06]) and dependency injection. Testing is conducted using an approach similar to ([FKK+05]) by reconstructing a *global execution* of a test run and then applying local and distributed *invariants* to that global execution. Simulated local executions are to be realized through the distributed middleware simulation environment MINHA ([CBCP11]). It should be pointed out that it is not the goal of this thesis to develop a fully fledged *e*GCS as this endeavour would far exceed the available

development time. The in part one outlined design considerations are applied to this prototype that strives to solve the immediate issue of extending the ADS laboratory group communication simulator so students can use their developed protocols in realistic scenarios. It is argued that the header-driven approach can provide a simpler composition model that is more familiar to students than the event-based interaction schemes of more complex (protocol) composition frameworks. For simulating protocol stacks MINHA may offer a simpler and more versatile approach than replacing layers of the stack with simulated counterparts though both models can be readily adapted using the proposed design.

3.4.2 Using the Java Programming Language

The first design decision covers the programming language that is considered for developing the prototype educational group communication system (*e*GCS). Currently the simulator provided in the ADS laboratory is implemented in Java, however the GCS toolkit *Spread* which is used for later tasks employs a *client-daemon* model where only the lightweight client has a Java implementation and the actual GCS daemon is based on C/C++. Such a GCS design, while offering a simple to use API and an easy adaptation for other programming languages on the client side, inhibits easy modifications or extensions to the underlying protocol stack and is therefore not particularly suited as the basis for an *e*GCS. For targeted group communication systems that could be modified or a self-developed GCS the Java programming language is primarily considered because it is used in introductory programming courses at the Vienna University of Technology. It also offers good platform independence with Java virtual machine (JVM) implementations for a wide range of target systems. Many other modern GCSs such as *Appia* and *JGroups* (see 2.3 Group Communication Systems and Protocol Composition Frameworks) are based on Java and within the covered works in this thesis the Java programming language was often used. Finally, the intended method of simulating the GCS stack, namely using the MINHA framework (see section 2.4.5) that is based on Java bytecode instrumentation, requires that both the (*e*)GCS as well as any student developed components and referenced libraries are implemented in Java.

3.4.3 Developing a New *e*GCS Prototype

Part one of this thesis has discussed approaches towards the development of an *e*GCS and it was concluded that currently there exists no single tool that can meet all the desired functionality. The covered non-educational group communication system and composition frameworks that are based on the Java programming language could in principle all be used as a basis or as components for the ADS laboratory *e*GCS prototype. At a first glance the GCSs *Appia* and *JGroups* appear to be good candidates for such a basis. *Appia* offers both a (generic) composition framework as well as a fully functional group communication stack with a modular design and contains a range of different protocol implementations. *JGroups* also gives developers the ability to modify the GCS protocol stack and create their own protocol compositions though it does not include a generic composition framework as is the case with *Appia*. Both GCSs are actively being maintained and extended and their source code is openly available[44]. Upon closer inspection however the complexity of either tool seems too inhibitive for the relatively short amount of time that is available in the ADS laboratory.

[44]The goals of this work necessitate that the source code for any candidate GCS is available and under a software license where it is permissible to make modifications and possibly publish these modifications for non-commercial use.

Other works that do not offer an entire (view synchronous) GCS implementation but can assist in the creation of complex protocol stacks and are particularly suited for developing an *e*GCS are *Neko* and *Kompics* (see Neko and Kompics in section 2.3.2). Neko provides an environment for developing and prototyping new protocols and protocol stacks and allows developers to easily switch between simulated and regular executions. There exist a number of reliable multicast protocols, failure detectors, and preliminary implementations of different forms of Consensus that can be drawn upon. Efforts have also been made to adapt the *nam* ([EHH+99]) animation library so it can visualize execution traces. In respect to the previously mentioned GCSs the Neko framework incurs a lot less learning overhead for students, however it is still rather complex and primarily geared towards research and not an educational scenario. Kompics on the other hand is a composition framework that was explicitly developed to reduce the complexity for developing distributed software. It has been successfully employed at the KTH Royal Institute of Technology in an advanced distributed systems courses where it replaced Appia as a composition framework ([AH08]). Kompics also allows users to switch between simulation and real execution by using a simulated network component in the composition through which all communication is routed. At the time of choosing a concrete design for the prototype the author was unaware of Kompics, which presents itself as an interesting basis for developing an *e*GCS or other educational tools for advanced distributed systems topics. However another approach that also promises to simplify protocol composition and offers an alternative to event-based interaction schemes is the header-driven model ([BMN05],[Men06]) covered in section (2.3.2). It was felt that developing a lightweight composition framework from scratch using such a header-driven approach could produce a tool with less learning overhead than a solution using frameworks like Neko as a basis. The following section highlights this design choice in more detail.

3.4.4 Primary Component Group Membership

After having defined the composition model the focus is placed on the prototype *e*GCS. Part one has outlined the basic building blocks of group communication (see chapter 2.2 Fundamental Concepts) and one essential design decision is whether to support a *partitionable membership* model or enforce *primary component group membership*. Many of the covered GCSs such as Appia, JGroups and Spread employ partitionable group membership. The advantage this brings is that partitions can be allowed to make progress, making them well suited for deployment scenarios in more unstable networks such as wide area networks (WANs) like the Internet, where partitioning is a more common occurrence. On the other hand primary component group membership can offer guarantees such as a single agreed upon sequence of views for which virtual synchrony applies without requiring additional protocol components to consolidate partition merges and determine a valid transitional set of processes that are in virtual synchrony[45]. Either membership variant can be implemented using the proposed header-driven composition framework so the question that is put forth is rather which group membership model is more appropriate for the ADS laboratory and an educational context? The previous section highlights that PCGM may be better suited for this task as the complexity of protocol development but also that of the GCS itself is somewhat reduced. In ([Sch02]) Schiper illustrates how straightforward it is to implement primary component group membership based on top of Consensus by presenting a simple algorithm consisting only of a few

[45]Partitionable group membership can emulate PCGM if non-primary partitions are not allowed to make progress.

lines of code (see PCGM and Consensus in section Error: Reference source not found). Students can be tasked to implement a Consensus protocol such as Chandra-Toueg Consensus ([CT96]) or Paxos Consensus ([Lam98]) and are then able to readily build a group membership service and other elements such as protocols supporting virtually synchronous communication themselves using this Consensus module.

3.4.5 Testing and Simulation

Potential approaches towards Simulation and Testing

Testing and performing simulations of the protocols and components forms an integral part not only in an educational scenario but for group communication systems in general. So far the design proposal has mainly discussed aspects related to the creation and composition of (GCS) components while ignoring how to simulate and test them. The initial situation of extending a group communication simulator to allow its protocols to be used in realistic scenarios is reversed as we now have to consider how to simulate “regular” student protocols developed in the *e*GCS prototype. Part one of this thesis covers several different approaches for simulating and testing GCSs (chapter 2.4) and outlines their advantages and disadvantages. A test approach based on reconstructed global executions similar to that described by ([FKK+05]) and two general simulation approaches are considered, namely replacing components of the stack with simulated counterparts and using the MINHA framework (see 2.4.5).

One important difference to the present group communication simulator is the additional timing assumptions required for protocols relying on execution modes of partial synchrony, introduced either through their own reliance on time or by using a failure detector abstraction. Currently the group communication simulator executes asynchronously and students must not make any such timing assumptions within their protocol code. Explicit use of timing is a potential source of problems when testing on different systems and introduces an element of non-determinism, because aspects such as thread-scheduling and the current processing load of the system can influence processing times. At the same time it more closely resembles regular testing approaches that are not conducted using discrete event simulations. To alleviate these timing issues and provide more control over a test's execution only local testing is currently considered where tests are able to draw upon the global knowledge of all stack instances. Using this approach may weaken or even avoid timing problems as actions can be initiated in relation to the global system state rather than being based on the progression of time.

Reconstructing a Global Execution and the Definition of Invariants

At the heart of the testing approach is the definition of *test scenarios* with global and local *invariants* that are applied to reconstructed *global executions* which follows the general approach described by Farchi et al. ([FKK+05]). A test scenario gathers relevant information from all stack instances during the test's execution which is then combined to reconstruct a consistent, time-ordered global execution. Based on this global execution the defined invariants are checked and a test result is generated accordingly. The described testing approach itself does not necessitate a simulation environment and can also be used in regular distributed environments. However a distinct advantage that can be gained through simulating such distributed executions locally is the potential

access to the global system state during execution. With this global knowledge the reconstruction of a global execution becomes trivial where otherwise local time differences between instances need to be accounted for when recording events. Furthermore providing a test scenario with global knowledge of all processes and their states during execution gives developers more freedom in setting up test conditions and lets them exert better control over the behaviour of the test. For instance a process may only start multicasting messages when a certain number of other processes have successfully joined the group. Alternatively if the simulation model or API of the protocol stack permits it we may intentionally delay or remove messages from the communication channel (or receive buffer of the transport protocol layer) to cause unfavourable conditions that are aimed at testing the robustness and correctness of the protocol implementation and its specified guarantees. If events such as the multicasting of a message are not tied to conditions of the global or local execution state their occurrence time during the simulation will need to be specified as neither of the considered simulation models operates asynchronously.

Invariants are to be defined by students to ensure that the targeted protocols meet their specification and are applied to the sequence of events contained in reconstructed global executions. Farchi et al. give the following example for a local invariant:

“view identifiers are monotonically increasing in absence of process crashes” ([FKK+05])

and the following examples for distributed invariants:

“(1) Membership layer guarantees: all view members agree on the same view members, id, and leader. Verification of this property provides an indication whether the occurred chain of view changes is legal or not. (2) Virtual synchrony, reliable delivery and Self-delivery guarantees - see Section 2.1. (3) Eventual process join and removal: If a new process has connected/disconnected to/from a group, a new view that includes/excludes this member is eventually established.” ([FKK+05])

Simulation of Protocol Layers in the Prototype *e*GCS Stack

First a simulation approach is considered where layers of the protocol stack are replaced by simulated counterparts. The most abstract simulation model of the prototype *e*GCS can be achieved by replacing the upper protocol layers that provide view synchronous communication primitives with simulated counterparts. In ([DH12a]) Drejhammar and Haridi outline the advantages of simulating view synchrony within a discrete-event simulation over implementing one of the several known algorithms on top of a network simulation, namely that it reduces complexity and exposes the application to all permissible timing behaviours under the model and not just those characteristic for the particular implementation. If the simulated VSC layer is adapted to conform to the abstract protocol type (of the regular VSC protocol) used in the prototype *e*GCS it can trivially replace its regular counterpart. This simulation model is well suited for early prototyping where precise performance metrics or accurate concurrent behaviour are less critical. Simulating lower level communication protocols or using a network simulator results in a model similar to those employed in Neko and Kompics (see 2.3.2) . In this case the behaviour of the stack observed during simulation runs is more representative. An issue that is encountered with this approach regardless of the position of the simulated protocols and components within the stack has to do with timing. Either the simulation is conducted in real-time, in which case the drawbacks of *network emulation* apply, or the

regular components of the stack need to be synchronized in respect to simulation time. To resolve the issue of synchronization bytecode instrumentation may be employed to replace code relying on system time so it is synchronized in respect to the simulation as is done in Kompics. Protocols relying exclusively on a failure detector abstraction or membership service instead of making explicit timing assumptions are also readily adapted by replacing these components with simulation-aware or simulated counterparts. It is also possible to require that any timing related calls are handled by specific libraries or an API that can then be switched between different execution modes, an approach that can be found within the Neko framework by relying on *NekoThreads*.

Simulation using the MINHA Framework

Of particular interest is the MINHA framework ([CBCP11] see 2.4.5) that offers the ability to transform regular Java code so it is executed in a simulated environment through bytecode instrumentation. This approach promises many advantages because it requires no modifications to the protocol stack for simulated executions and provides a realistic outlook on its performance and general behaviour (such as concurrency). MINHA presents a novel route that could also lead to a simplification in the development of other educational tools. The framework simulates multiple concurrent Java Virtual Machines (JVMs) and their interconnection through network sockets in a single JVM instance by using bytecode instrumentation. This reduces the simulation overhead of using multiple JVMs but more importantly many of the issues encountered when considering the simulation of GCSs are also addressed.

Firstly the in the previous subsection mentioned synchronization issues that can be encountered when simulating parts of the protocol stack are removed. MINHA *transparently* transforms the targeted program code so it is integrated into the simulation model and executes in simulation time. This removes the necessity for providing specific libraries or real-time capable simulated (emulated) protocol layers and components, thereby affording a better separation of concerns and simplifying the design. Secondly, MINHA is aimed at providing a realistic simulation environment. It is

“a system that virtualizes multiple JVM instances within a single JVM while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of the actual distributed system.” ([CBCP11])

In contrast to most abstract simulation environments MINHA can offer more accurate representations of how protocols and the entire GCS stack will behave once deployed on a real distributed system. This gives learners the ability to experience and debug a wider range of issues that otherwise might not be encountered. Thirdly, the framework does not dictate the use of specific Java libraries or a particular design approach. As long as a library does not interfere with the rewriting mechanism or is incapable of being transformed it is “compatible” with the simulation environment. It is hence also possible to perform simulations of compositions created with other frameworks such as Neko or Kompics[46] as well as Appia, JGroups and other modular GCSs based on Java. Educators and learners are thereby given the freedom and flexibility to draw from a wide range of tools while retaining the simulation capabilities offered by MINHA.

[46]The compositions would need to use the “regular” mode to avoid executing as a simulation within MINHA.

Design Decisions for Simulation

The header-driven composition model affords an easy reconfiguration of compositions and can render the specification of protocol stacks more straightforward and less error prone. If lower level networking components are replaced by simulated versions a model similar to Kompics or Neko could be achieved. On the other hand if the GMS and view synchronous communication layer are simulated it would result in an approach like that of Drejhammar et. al ([DH12b]). In both of these cases however it must be ensured that the notion of time is consistent amongst simulated and regular components, requiring that either the regular execution be synchronized in respect to simulation time or that the simulated components are real-time capable. Alternatively the entire composition can be executed on multiple virtualized JVMs in simulation[47] time using the MINHA framework without the necessity to replace any regular components of the protocol stack with simulated counterparts.

For the *e*GCS prototype a simulation approach using MINHA is chosen as it promises to simplify the creation and execution of simulations and potentially offers more flexibility than the method of replacing protocol layers in the stack with simulated counterparts.

3.4.6 Putting it all Together

From the above individual design aspects the following overall design emerges. A lightweight protocol composition framework using a header-driven composition model and dependency injection to compose protocol stacks serves as the basis. On top of this framework a basic primary component group membership service is implemented using Consensus as a basic building block. Students will receive a skeleton composition of the GCS stack with placeholder components for the parts that need to be implemented such as (view synchronous) multicast protocols with different ordering and delivery guarantees. Due to the limited time available in the ADS laboratory it is not feasible to require that students design and implement large portions of the GCS. Testing and performance evaluations are conducted through the MINHA framework in a similar manner to how the current group communication simulator ([FG11]) uses *TestProviders.* This is achieved by requiring students to specify test scenarios where the targeted protocol stack is executed on multiple simulated and networked JVM instances. Students need to define the external events that influence the simulation such as failures and when method invocations on individual stack instances (processes) take place. Furthermore they have to specify invariants that are applied to the reconstructed global execution of the test scenario's execution based on logged information which determine if the protocol behaved correctly or not. The following Figure (Fig. 27) illustrates the design and how test scenarios rely on the same protocol stack and composition model for testing purposes.

[47] Hence providing *synchronized network emulation.*

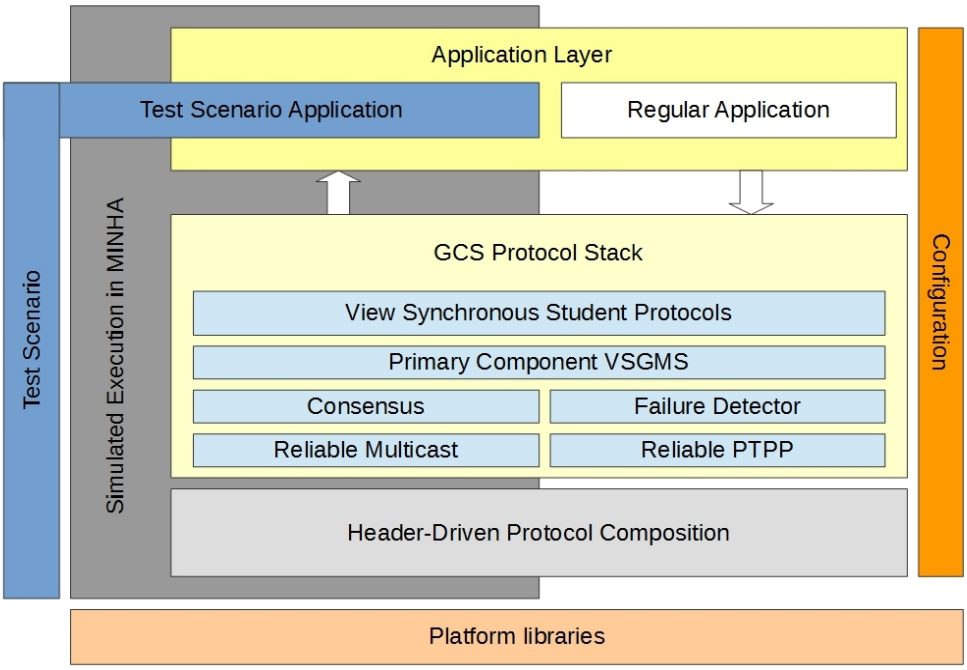

Fig. 27: Proposed Architecture of the eGCS prototype

3.5 Prototype Development

3.5.1 Implementation Overview

The following sections describe implementation details for the prototype *e*GCS based on the general design proposal presented in the previous chapter (3.4). A primary goal was the development of a functional GCS protocol stack based on the the header-driven composition model (see Header-Driven Protocol Composition in 2.3.2). The implementation serves to test the feasibility of the header-driven approach in respect to the educational setting outlined in this part of the thesis. The design process was iterative and focused on reducing the programming and learning overhead required to implement protocols and protocol stacks in the model. The prototype *e*GCS relies on Consensus as a basic building block and employs a modified version of the Chandra-Toueg Consensus algorithm (see 3.5.3). Both the group membership service and view synchronous multicast are provided through a combined protocol that uses this modified Consensus to agree upon both the next membership set as well as the messages seen during the previous view. Initial experiments using the MINHA framework for simulation and testing were conducted, revealing that while MINHA can be a powerful tool for more experienced developers the framework is currently not ideally suited for an application in an educational scenario. As outlined in the previous chapter the replacement of protocol layers with simulated counterparts is discussed as an alternative approach. Aside from a first functional *e*GCS prototype a second iteration of the design was made with a stronger focus on simplifying the API and a more strict approach towards the header-driven composition model that is still in a prototypical development phase. Part of the *e*GCS prototype

development also involved implementing the modality of the original ADS laboratory assignment with its static group and asynchronous communication model as a use case and to highlight the functionality and flexibility of the proposed approach. This implementation is fully functional and solves the initial goal of allowing student protocols to be used in realistic scenarios while also providing facilities for simulation and testing (see 3.5.7).

3.5.2 Header-Driven Composition and Dependency Injection

Description

Chapter (2.3) provides an overview of protocol composition frameworks and their role in developing group communication systems. From a practical standpoint it is preferable to use frameworks and composition models with as little learning overhead as possible, even if this results in less flexibility and functionality. Students may only have a very limited amount of time for assignments and placing on them the burden of familiarizing themselves with complex tools is undesirable. One possible approach to render an underlying complex composition framework more transparent and manageable to students is to provide a partially implemented GCS where only specific placeholder components require the addition of protocol logic. Unfortunately this does not resolve the issue of encouraging learners to engage and experiment with the GCS and composition framework beyond the given task in a constructive manner and the complexity of the framework itself is not reduced. As is later argued the header-driven approach may present a compositional model that can greatly reduce this learning overhead.

Most protocol composition frameworks and modular GCSs rely on a general-purpose event-driven model to manage protocol interactions ([BMN05]). Such event-based interaction mostly has a loose coupling between components and can allow for a very modular design. However more complex protocol stacks such as that of an entire GCS can quickly lead to complex code and difficulties in regards to correctly binding the event handlers of components to channels or event queues and routing events to the correct components. It has been argued that using such an event model may not be an ideal scheme for protocol composition (see for instance [BMN05],[Men06][Rüt09],[RWS06]) and that protocol stacks are actually tightly coupled in the downwards direction ([BMN05]). An alternative approach to protocol composition that greatly reduces the architectural overhead found in event-driven models was proposed by Bünzli et al. (see Header-Driven Protocol Composition subsection in 2.3.2) which uses message headers rather than events as the basic driving mechanism for composition. At the core of this model is the idea of using header-handler pairs where headers are uniquely named, *typed* containers for data and the handler is exclusively bound to its corresponding header. Messages are composed of a stack of headers where headers are pushed onto the stack by each protocol layer as the message traces a path through the composition until it is *remotely dispatched* (sent) to its destination. A header handler, where header and handler pairs are registered, is responsible for dispatching incoming messages to the right handler. This is done by performing a lookup for the respective handler of the topmost named header and then passing it as well as the entire message to that handler for further processing. Upon completion, the handler usually re-dispatches the message which will result in a call to the handler of the next header on the stack. In this manner a message traces the reverse path it took based on the sequence of headers on the stack without having to explicitly bind handlers to channels or events. The header-driven

approach provides a composition model where a lot of the "wiring" of components, namely specifying the upwards channel of messages, is implicitly handled through defining header-handler pairs. A protocol composition framework based on the header-driven approach promises to reduce complexity and seems ideally suited for the task of developing an *e*GCS prototype. Another interesting aspect that comes into play with the aforementioned approach is when we consider individual protocol layers and the specification of their downwards dependencies. If these dependencies are expressed as abstract protocol definitions the (constructor) *Dependency Injection* design pattern ([Fow04]) seems like a natural choice for composing protocol stacks. This *Inversion of Control* (IoC) design pattern is well suited for a composition framework and can help simplify testing. The proposed design for the *e*GCS prototype will therefore employ the header-driven model and dependency injection for composing protocol stacks.

Protocol Creation and Composition

As outlined previously (3.4) the proposed design for the composition framework is based on a header-driven approach and the dependency injection (DI) design pattern ([Fow04]). Constructor dependency injection lends itself well for composition and testing where protocols and components are described through abstract implementations or interfaces. For the instantiation of protocol stacks an earlier iteration of the prototype employed the PicoContainer library ([Pic11]). Basic testing indicates that PicoContainer does not interfere with the bytecode instrumentation process of the MINHA framework that is to be used for simulation purposes. It was however decided that for the limited scope of developing this *e*GCS prototype its inclusion adds unnecessary complexity and the prototype can later be readily extended to rely on it or other, more heavyweight dependency injection frameworks. In spite of using a DI framework many of the dependencies of protocol modules generally need to be explicitly defined rather than allowing the framework to automatically inject components that match the specified abstract dependencies. In part this problem arises from the issue of how to effectively model an aggregation of properties, in part the question is raised how a dependency injection framework could automatically choose the most suitable instance. Consider an interface describing an abstract unicast protocol for point-to-point communication. A composition may contain two concrete protocols implementing this interface, one offering *quasi-reliable* communication while the other provides unreliable lossy links. A third protocol, say a simple messaging system, may only specify this abstract unicast protocol as a lower dependency without any additional guarantees. In this case it is up to the composer to decide if the service should support the reliable delivery of messages or if it is acceptable that messages may be lost. An interesting question that also arises is how this simple messaging protocol instance using the reliable unicast link as a lower dependency should be treated. The reliable property of its lower protocol may now also render the messaging protocol itself reliable while other properties such as the prevention of delivering duplicate messages of the lower layer may not be enforced on the layer above. Clearly, the development of protocol composition frameworks that also consider the aggregation of such properties for automatic dependency injection is a complex and difficult problem that is well beyond the scope of this work.

Protocol composition using the header-driven approach is currently based on a prototype implementation that follows the general guidelines presented by Bünzli et al. ([BMN05],([Men06]). The abstract generic class *Handler* with an upper type bound of *java.io.Serializable* serves as the

base for all header-handler pairs. A *Header* class in the same package provides a container for this typed data and is instantiated through calling the defined method *invoke*. Type checking is done statically by the compiler as Java employs type erasure during compilation. The resulting Header object also holds a unique name that identifies the corresponding Handler, so that the header and its contained parameter (and the tail of the message) can be dispatched to the correct Handler at the remote location. Handlers can only be instantiated in the context of a *HeaderHandler*, which is also *named*. The HeaderHandler creates a root namespace for Handlers[48] and implements the *dispatch* method for dispatching a message with its Header stack to the next Handler, hence forming the base component for protocol compositions. HeaderHandler instances hold the references to all Handler instances created in their context and enforce a unique name at construction time (constructing a new object that extends the type Handler will fail if another Handler object is already registered under that name). The Handler also specifies an abstract method *handle* that takes as parameters the typed container and the message continuation retrieved from the Header. This model weakens the in ([BMN05]) described design because Handler creation is possible at runtime (only unique naming is required). Furthermore the property of symmetrical protocol stacks is not enforced and it is up to protocol developers to choose a reliably unique naming convention for the Handlers they instantiate. While the current implementation will prevent two Handlers to exist with the same name within the namespace of a single HeaderHandler, no guarantees are given across different compositions. Two protocol stacks, S_a and S_b with the same name may each contain Handlers using the same name but represent different implementations, possibly even differently typed. S_a and S_b can be said to be incompatible stacks and a remote dispatch from one to the other needs to be prevented. The naming convention for stacks and Handlers however should in itself prevent this situation from occurring as programmers following the guidelines must not give two different stack compositions or Handlers the same, supposedly unique, name. In this context another issue that needs to be addressed is the bootstrapping and synchronized start of protocols and protocol stacks. It is assumed that the entire stack is first assembled and then started, however no mechanisms have been put into place to prevent protocols from being dynamically added during runtime. Clearly, this could cause message dispatch to fail if the addition is not synchronized for all locations, as handlers might not yet be registered.

Summing things up, to create a new protocol that uses the header-driven approach at least one class extending Handler needs to be implemented and instantiated with a unique name and the reference to the corresponding HeaderHandler as parameters. After the transmission of a message over the network the remote dispatch will only occur if another Handler with the same unique name is registered. It is up to the developer to ensure a unique naming scheme and symmetric protocol stacks.

[48]Generally a Handler will be created in the context of a (uniquely named) Protocol, where a naming scheme of the form *protocol.handlername* may be used, however as long as the Handler's name is unique and a handler Object is present at all stack instances dispatch will succeed.

```
1       class MyHandler extends Handler<MyType> {
2
3               private Counter c;
4
5               public MyHandler(HeaderHandler hhandler, String uname,Counter c) throws
6               DuplicateHandlerException, HeaderHandlerClosedException {
7                       super(hhandler, uname);
8                       this.c = c;
9               }
10
11              @Override
12              public void handle(MyType params, HeaderBasedMessage tail) {
13                      c.increment(params.getCount()); // do computation
14                      dispatch(tail);         // re-dispatch the message so the
15                                              // remaining header stack can be processed
16              }
17      }
```

Fig. 28: Example code of a class extending the abstract class Handler using MyType as a container for parameters of the deferred method invocation.

The above figure (Fig. 28) gives an example of how a header-handler pair is implemented in the current protocol composition framework. The *handle* method serves as a callback that the HeaderHandler calls and to which it passes the correctly typed parameter container and the tail of the message with the remaining Header stack. Information passed through the typed container is then used to update the protocol state (in this case incrementing a counter using the provided value). Extensive computations in the *handle* callback should be avoided. Finally the tail of the message containing the remaining header stack is re-dispatched. Messaging protocols may also buffer this tail until the provided guarantees are met or discard the message (for instance in case of duplicates) by not calling dispatch. Figure 29 (Fig. 29) illustrates how a protocol can add a deferred method invocation in the form of a Header to the message before passing it on for further processing. Using the method *invoke*, defined in the Handler base class, together with a parameter of type *MyType* returns a Header that can then be added to the header-based message. In this particular case the *MyType*[49] class and the “myhandler” Handler instance refer to those of the previous example (Fig. 28).

```
1       public void send(HeaderBasedMessage message,int c) throws Exception {
2               MyType increment = new MyType(c);
3       // invoke a deferred computation e.g. create a header that contains the
4       // increment parameter and add it to the header stack
5       // NOTE: the compiler does static type checking for myhandler which
6       // is parameterized with MyType e.g. it is an instance of Handler<MyType>
7               message.add(myhandler.invoke(increment));
8       // pass the message to the lower protocol layer
9               lower.send(message);
10      }
```

Fig. 29: Example code of how a deferred method invocation is added to a Message in the form of a Header.

The composition of protocol modules is conducted as would be the case with regular object instances using a dependency injection design pattern. Interfaces and abstract classes are used to define dependencies (such as lower level protocols) that are exposed in the object's constructor. The (incomplete) example code from (Fig. 30) highlights how a simple GCS stack may be composed. *StringMessagingProtocol* acts as a front end where text messages can be sent using FIFO-ordered

[49]MyType is used to highlight the use of self defined type containers. Parametrizing the *MyHandler* implementation with an *Integer* would of course make more sense for this particular example.

view synchronous multicast. The VSGMS protocol responsible for group membership and basic view synchronous multicast communication relies on a *ConsensusProvider* for creating new Consensus instances, a failure detector and reliable multicast and unicast communication.

```
1       // new headerhandler with namespace uname
2       HeaderHandler handler  = new HeaderHandler(uname);
3       // transport for remote dispatching
4       BasicTransport  tp = new BasicTransport(handler,"BasicTransport", ... );
5       DiamondPFailureDetector fd = new DiamondPFailureDetector("DiamondPFD", tp, ... );
6       // the DiamondP HeartbeatFailureDetector also serves as a unicast protocol
7       HeaderBasedProtocol unicast = fd;
8       PingProtocol ping = new PingProtocol("PingProtocol", unicast);
9       ReliablePTPP ptpp = new ReliablePTPP("ReliablePTPP", ping);
10      HeaderBasedMulticastProtocol mcast = new ReliableMulticast("ReliableMulticast", unicast);
11      // Consensus provider for getting Consensus instances
12      ConsensusProvider cs_p = new HeaderBasedCTConsensusProvider("CS_Prov", fd, unicast, mcast);
13      String groupname = "mygroup";
14      VSGMS vsgms = new VSGMS(handler.getName()+"VSGMS."+groupname, fd, cs_p, mcast, unicast);
15      ViewSynchronousCommunication vsc = vsgms;
16      GroupMembershipService gms = vsgms;
17      ViewSynchronousCommunication vsc_fifo = new VSCFIFO("VSC_FIFO", vsc);
18      StringMessagingProtocol smp     = new StringMessagingProtocol("Smessenger",vsc_fifo);
19      // start accepting dispatched messages
20      handler.start();
21
22      Future<? extends View> joined = gms.join();
23
24      joined.get(2l, TimeUnit.SECONDS);
25      // sends a message using view synchronous fifo multicast
26      smp.send("hello world");
27
28      String answer = smp.receive();
```

Fig. 30: Example code of how a GCS protocol stack may be composed

3.5.3 Group Communication System

GCS Overview

At the heart of the prototype *e*GCS design is the actual protocol stack responsible for providing a group membership service (GMS) and virtually synchronous multicast communication (VSC). As outlined in the design proposal (3.4) only *primary component group membership* is currently considered. The iterative design process renders it difficult to formulate a formal specification of the GCS beforehand as different approaches can have a direct effect on the specification. Properties such as open or closed groups, the form of agreement used for group membership and virtual synchrony, as well as guarantees such as sending view delivery or same view delivery are all influenced by the particular design. Once more it has to be highlighted that the current prototype is geared towards experimenting with different GCS designs that may be used in an educational context. In this particular case ease of use and simplicity are in the focus, however correctness is still a central requirement. At this stage of development a formal specification of the GCS is not made, yet once a final design is chosen this will have to be done.

Currently the GCS implementation relies on Chandra-Toueg (CT) Consensus as a basic building block that is extended with similar concepts as ([HMRT99,GHRT01]), namely the ability of the coordinator of a Consensus round to delay sending a proposal once a majority of votes has been received and the ability to apply a function $\mathcal{F}$ to the set of estimates to generate a proposal rather than having to choose one of the initial estimates. Both the group membership service and view synchronous multicast are provided through a combined protocol that relies on Consensus to agree upon the next membership set as well as the set of messages delivered during the previous view. Alternative approaches such as separating GMS and VSC are discussed in the subsection on virtually synchronous communication (Combined GMS and VSC Protocol).

Group Communication System based on Consensus

In part one of this thesis the topic area of *group communication* is outlined in detail (2.2). The primary component group membership problem is an *agreement* problem that is solvable using *Consensus* with very little coding effort[50]. Other important problems in the context of reliable group communication such as total-order multicast can also be implemented using Consensus as a basis, in fact ([CT96]) show that total-order broadcast and Consensus are equivalent[51], where one solution can be used to solve the other. Approaches such as that of Greve et al. ([GHRT01],[GlN04]) hence use Consensus, or more precisely a generic agreement framework based on CT Consensus, as the basic building block for developing a primary component GCS. Another example of a primary component GCS that uses a modified CT Consensus for implementing virtual synchrony is *Phoenix* ([Mal96]). For the prototype eGCS an implementation based on Consensus can provide several advantages. Students can rely on the Consensus component when developing other protocols. They may also be tasked with implementing different Consensus algorithms that can then be used in the GCS protocol stack if the API is kept the same. The properties and guarantees of Consensus algorithms are also generally better specified than alternative approaches used for implementing group membership and virtual synchrony.

An astute reader will have noticed that both examples of GCSs that use Consensus as a building block rely on modifications to the Consensus algorithm. These modifications are made to account for the specific requirements when considering view synchronous communication (VSC). For instance in ([Mal96]) CT Consensus is extended for the *view change protocol* to wait until it has received an *estimate* from all processes that are not suspected. Regular CT Consensus will progress once a majority of estimates has been collected by the current round's coordinator. This modification does not violate the liveness properties of the original algorithm as the strong completeness of the failure detector ensures that every crashed process is eventually suspected, however agreement can be further delayed as the coordinator has to wait for slow processes that are not suspected. Another modification to “regular” CT Consensus can be found when looking at how a coordinator generates its proposal. In the case of Phoenix the coordinator sends a proposal that contains the union of all unstable message sets it has received through estimates, if none of these estimates is a proposal from a previous round as well as the new membership set that excludes processes based on its $\diamond\mathcal{S}$ failure detector. This means the coordinator's proposal is not one of the original estimates, a property found

[50]See Fig. 8 in (2.2.6 The Group Membership Problem and Virtual Synchrony) for a basic algorithm.
[51]Note that total-order (atomic) multicast has been shown to be harder than total-order broadcast, see [GS96]

in the original CT Consensus algorithm. Similarly Consensus instances with both of the described modifications can also be created using the *generic agreement framework* presented in ([HMRT99]).

Currently the prototype *e*GCS is also based on a modified implementation of CT Consensus that waits for estimates from all non-suspected processes and not just the minimum majority. Furthermore the Consensus instance relies on a so called *ProposalChooser* that defines a function on the set of estimates received by the coordinator to generate a proposal that is not necessarily part of the set of received estimates. The validity property of Consensus (If a process decides a value *v*, then *v* was proposed by some process, see 2.2.4) is a safety property to ensure that no unwanted actions can be taken. All participants of the modified Consensus instance need to agree upon the same *ProposalChooser* function and it should produce a predefined set of possible output results ([HMRT99]). Furthermore this function can only be applied by a coordinator if it has not received an estimate that is a proposal from a previous round[52]. The reason for allowing the application of such a function also ties in with another feature that can be found in the generic agreement framework ([HMRT99]), namely the ability of a process to change the value it has previously proposed (sent an estimate) under certain circumstances.

To understand why this approach is valid we need to take a closer look at the CT Consensus algorithm. CT Consensus is split up into four phases per round. During the first phase processes send their estimates to the coordinator of that round while in the second phase the coordinator waits for a majority of estimates, chooses a proposal from the set of estimates and then sends that proposal to all participants. Once the coordinator receives an acknowledgement to this proposal from a majority of processes (phase three) it reliably broadcasts said proposal as the decision (phase four). If a process suspects the coordinator of the current round it sends a negative acknowledgement for the proposal[53] and moves on to the next round where the same four phases are initiated. Coordinators must always choose their proposal from the set of estimates that has been the most recent proposal, indicated by the most recent round number provided with the estimate (initial estimates have a round number of 0). Valid proposals have to be “locked in” by processes to ensure that every process decides on the same value, even if agreement is formed concurrently in more than one round[54]. A process that has not received such a proposal from the coordinator, subsequently suspected it, and moved to the next round will still have an estimate with round number 0. Since a coordinator can only lock in a proposal if it receives a majority of acknowledges and can only determine a proposal once it has at least a majority of estimates there must always be at least one process from the previous round with an updated estimate with a number greater than 0. Hence a process with an estimate of round number 0 cannot affect a valid round by changing its proposal. The only time that new modified proposal may be considered is when there has not yet been a coordinator able to make a proposal and all estimate round numbers are still 0. Both changing an initial estimate or applying a function to the set of estimates by the coordinator fall under the same category. They are only permissible if no

[52]Otherwise there is the possibility that Consensus in a previous round may decide on a different proposal.

[53]Interestingly the original algorithm only specifies the sending of a negative acknowledgement in phase three while waiting for an estimate from the coordinator and states nothing about waiting for the coordinator to broadcast the decision. Hence processes actually move to the next round and send their estimate to the new coordinator once they complete phase three unless they receive the decision beforehand.

[54]In CT Consensus it is conceivable for a coordinator to decide and be suspected by processes, yet its reliable broadcast of the decision is merely late. If a new decision were formed in a consecutive round with a different value processes may disagree invalidating the *Agreement* property of Consensus. Hence coordinators must always use the most recent previous proposal instead of proposing something new. See [CT96] for more details on the algorithm.

other proposal is known because then it can be guaranteed that correct processes agree on the same decision.

In the context of the ADS laboratory and the prototype *e*GCS the interesting question arises if the design should aim for the implementation of a generic agreement framework as in ([HMRT99]) or if regular Consensus is sufficient. The former approach provides a more flexible form of Consensus however it may be more challenging to adapt other Consensus approaches to conform with the extended API. Using only basic Consensus primitives on the other hand may increase the complexity of other protocols and impact performance as additional rounds of communication are required. A generic Consensus service appears to be the more promising route at this point and time and hence will be used in future iterations of the (*e*)GCS design.

The following pseudo code (Fig. 31) outlines the modified CT Consensus algorithm ([CT96]) currently used for the GCS. Lines 18 and 19 are different from the original code which only waits for the minimum majority ($\lceil (n+1)/2 \rceil$) before the coordinator sends its proposal to all participants. The current design that is also found in ([Mal96]) lets the coordinator collect all estimates from non-suspected processes but can only progress if it also has more than a minimum majority ($\lceil (n+1)/2 \rceil$) to ensure safety. Liveness properties are not affected by this change because the utilized failure detector exhibits *strong completeness*. Lines 22 and 23 show a modification that is also described in ([HMRT99]) where the coordinator can apply a function $\mathcal{F}$ (the *ProposalChooser* class implements this functionality in the code) to all received estimates of that round in order to generate a proposal. This action is only permissible if no other proposal from a different Consensus round has been observed by the processes partaking in the current round of Consensus. Finally line 42 shows a slight modification to avoid unnecessary message communication. In the original Consensus algorithm ([CT96]) processes that have sent their acknowledgement or negative acknowledgement in *phase 3* will move directly to *phase 1* of the next Consensus round and send their estimate unless a decision reaches them. By blocking until either a decision is received or the coordinator is suspected we can avoid processes starting a new round of Consensus. Again because of the *strong completeness* of the used failure detector ($\diamond\mathcal{S}$) liveness is not affected and every correct process will eventually suspect a failed coordinator and move to the next round of Consensus or receive the decision through reliable broadcast.

```
1   procedure propose(v_p)
2       estimate_p ← v_p                    {estimate_p is p's estimate of the decision value}
3       state_p ← undecided
4       r_p ← 0                             {r_p is p's current round number}
5       ts_p ← 0                            {ts_p is the last round in which p updated estimate_p , initially 0}
6
7       {Rotate through coordinators until decision is reached}
8
9   while state_p = undecided
10      r_p ← r_p + 1
11      c_p ← (r_p mod n) + 1               {c_p is the current coordinator}
12
13      Phase 1:    {All processes p send estimate_p to the current coordinator}
14      send (p, r_p , estimate_p , ts_p ) to c_p
15
16      Phase 2:    {The current coordinator gathers ⌈(n+1)/2⌉ estimates and proposes a new estimate}
17      if  p = c_p then
18      wait until[for at least ⌈(n+1)/2⌉ processes q : received (q, r_p , estimate_q , ts_q ) from q or q ∈ D_p ]
19          {At least a majority of estimates was collected}
20      msgs_p [r_p ] ← {(q, r_p , estimate_q , ts_q ) | p received (q, r_p , estimate_q , ts_q ) from q}
21      t ← largest ts_q such that (q, r_p , estimate_q , ts_q ) ∈ msgs p [r_p ]
22      if t = 0 then       {Apply function F to all relevant estimates to generate a proposal}
23      estimate_p ← F ( ∀ (q, r_p , estimate_q , t) ∈ msgs_p [r_p ])
24      else estimate_p ← select one estimate_q such that (q, r_p , estimate_q , t) ∈ msgs_p [r_p ]
25      send (p, r_p , estimate_p ) to all
26
27      Phase 3:    {All processes wait for the new estimate proposed by the current coordinator}
28      wait until [received (c_p , r_p , estimate_cp ) from c_p or c_p ∈ D_p ] {Query the failure detector}
29      if  [received (c_p , r_p , estimate_cp ) from c_p ] then        {p received estimate_cp from c_p }
30          estimate_p ← estimate_cp
31          ts_p ← r_p
32          send (p, r_p , ack) to c_p
33      else send (p, r_p , nack) to c_p                    {p suspects that c_p crashed}
34
35      Phase 4:    {The current coordinator waits for ⌈ (n+1)/2 ⌉ replies. If they indicate that ⌈ (n+1)/2 ⌉ }
36          { processes adopted its estimate, the coordinator R-broadcasts a decide message }
37
38      if p = c_p then
39          wait until [for ⌈ (n+1)/2 ⌉ processes q : received (q, r_p , ack) or (q, r_p , nack)]
40          if  [for ⌈ (n+1)/2 ⌉ processes q : received (q, r_p , ack)] then
41              R-broadcast(p, r_p , estimate_p , decide)
42      else wait until [state_p = decided or c_p ∈ D_p ]       {Query the failure detector}
43
44  {If p R-delivers a decide message, p decides accordingly}
45
46  when R-deliver(q, r_q , estimate_q , decide)
47      if  state_p = undecided then
48          decide(estimate_q )
49          state_p ← decided
```

Fig. 31: Modified CT Consensus algorithm with failure detector $\mathcal{D} \in \diamond\mathcal{S}$ and function $\mathcal{F}$ for choosing a proposal

Combined GMS and VSC Protocol

The previous subsection outlined how Consensus can be used a basic building block of a GCS. Now we will take a closer look at how the group membership service and virtual synchrony are implemented in the prototype *e*GCS using this concept. The *virtual synchrony* property (see 2.2.6) holds for processes *p* and *q* when both transition from the same view *v* to the next view *v+1* and agree upon the messages received in the previous view *v*. Virtual synchrony allows the implementation of reliable multicast protocols for dynamic groups, it "... guarantees the agreement property of reliable broadcast among the servers in a stable component" ([Cach08]). A group membership service on the other hand is used to agree upon the dynamically changing set of processes that make up the group by providing a succession of agreed upon *views* of the current membership (2.2.6). Since virtual synchrony is defined in the context of views both view synchronous communication and the group membership service are intimately linked.

If the group membership service is provided separately, a VSC protocol needs to be synchronized in respect to the views delivered by the GMS. Consider two instances of Consensus, one deciding on the membership set of the next view while the other is responsible for agreement on the messages delivered during the previous view. In itself Consensus will either guarantee agreement on a value or block if a majority of processes fails[55]. However in respect to two concurrently executing Consensus instances on the same set of processes temporal differences may let one instance reach agreement just before a majority of processes fails while the other remains blocked indefinitely. A solution may be to avoid this concurrency and block the initiation of Consensus for a new view until agreement has been reached on the message set of the current view. The obvious downsides to such an approach are that other services relying on the GMS would also be affected by this blocking and an additional instance of Consensus is executed for the VSC protocol.

A different approach to solving the issue of providing separate GMS and VSC protocols that does not rely on a Consensus abstraction is the *flush* protocol presented by Birman et al. in ([BSS91]) for the ISIS toolkit. In this case communication channels between processes are assumed to provide FIFO ordering guarantees and the protocol relies on a *perfect failure detector*[56]. During a view change with no failures when the GMS delivers a new view *i+1* all members of this new view send each other a "flush i+1" message and cease sending new messages until they have received a "flush i+1" message from every other process. Because of the FIFO guarantees of the communication channels if every process has received such a message from every other process and no failures occur during the flush protocol the virtual synchrony property is upheld. For situations where failures occur during a view change the authors describe the following solution:

"The solution to this atomicity and virtual synchrony problem is most readily understood in terms of the original n^2 message flush protocol. If we are running that protocol, it suffices to delay the installation of $view_{i+1}$ until, for some $k \geq 1$, flush messages for $view_{i+k}$ have been received from all processes in $view_{i+k} \cap view_{i+1}$. Notice that a process may be running the flush protocol for $view_{i+k}$ despite not yet having installed $view_{i+1}$." ([BSS91])

[55]Under the assumption of a long enough *stable* period of the network for Consensus to be possible.
[56]The successive view outputs of the GMS are used as an *AP* failure detector. (see 2.2.3 Failure Detection)

If failures occurred during a view change (a new view is delivered by the GMS before the previous view was installed) processes forward each other their unstable messages[57] from failed processes, mark these messages as stable and send a flush message for the new view. On the receiving side processes will discard any duplicate unstable messages received in this fashion. The same process of forwarding unstable messages can again occur if another process fails until eventually a view i+k will be delivered[58] where no failures occur and all pending views are subsequently successfully installed. Figure 32 (Fig. 32) is an illustration from ([BSS91]) that visualizes an execution with additional failures while flushing is in progress. The reader is pointed towards ([BSS91]) for a more detailed and formal description of the algorithm.

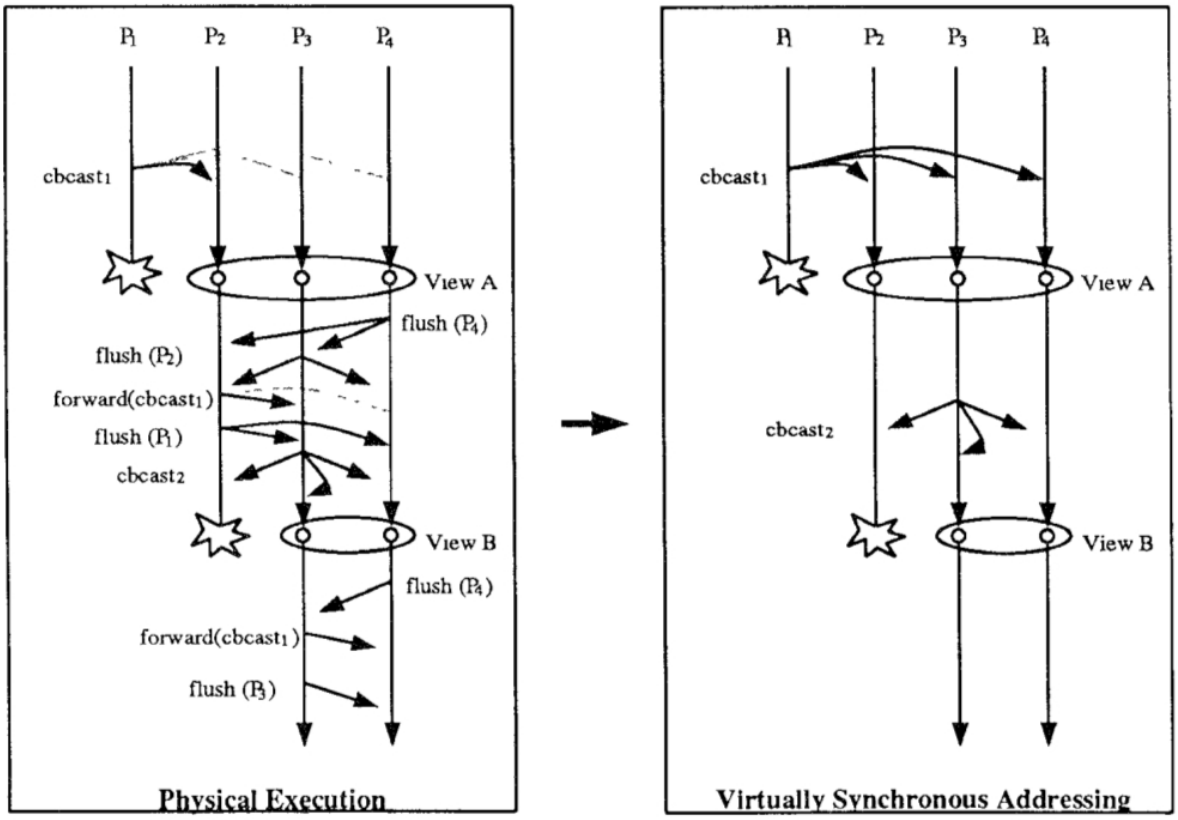

Fig. 32: Example of the flush algorithm with multiple process failures [BSS91]

The current implementation combines both GMS and VSC in a single protocol (hereon called VSGMS or *view synchronous group membership service*) using an instance of the previously described modified Consensus algorithm to agree upon both the next membership set as well as the set of messages that were delivered to all correct processes of the new view in the previous view. Therefore the issue where only one component is live is avoided for the VSGMS may either block entirely or safely progress to the next view with virtual synchrony ensured. The approach is conceptually similar to that of *Phoenix* with the difference that the VSGMS protocol draws upon a modified (generic) Consensus protocol instead of using a specific implementation. Implementing the

[57]Stable messages are agreed upon by all members of the current view and hence need no further confirmation.
[58]ISIS uses program controlled crash so the group may perform collective suicide. Other forms of GMS may block indefinitely if too many failures occur and no new view can be installed.

VSGMS based on the modified Consensus protocol is straightforward, however other protocols that only need to rely on a GMS and do not require view synchronous communication primitives are affected by the slower responsiveness of the VSGMS and longer intervals during which the protocol is blocked. The implementation provides *sending view delivery* of messages and hence requires the application to eventually block the sending of messages during view changes. If only *same view delivery* delivery is required a simple FIFO buffering scheme allows the application to continue multicasting messages during a view change (see [CKV01] for more details). Future implementations of the eGCS prototype may employ the *flush* protocol so separate GMS and VSC protocols can be provided. The VSGMS is characterized by the following properties (definitions from [CGR11],[CKV01]), a more detailed specification of the finalized prototype will have to be made.

VSGMS1: *Validity:* If a correct process *p* multicasts a message *m*, then *p* eventually delivers *m*.

VSGMS2: *No duplication:* No message is delivered more than once.

VSGMS3: *No creation:* If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

VSGMS4: *Uniform View Agreement:* If some process installs a view V = (id, M) and another process installs some view V' = (id, M'), then M = M' .

VSGMS5: *Primary Component Membership:* There is a one-to-one function *f* from the set of views installed in the trace to the natural numbers, such that *f* satisfies the following property. For every view V with *f* (V) > 1 there exist a view V' , such that *f* (V) = *f* (V') + 1, and a member *p* of V that installs V in V'.

VSGMS6: *Completeness:* If a process p crashes, then eventually every correct process installs a view (id, M) such that p ∉ M.

VSGMS7: *Accuracy*: If there is a time after which processes p and q are alive and the channel from q to p is up, then p eventually installs a view that includes q, and every view that p installs afterwards also includes q.

VSGMS8: *Virtual Synchrony:* If processes *p* and *q* install the same new view *V* in the same previous view V´ , then any message delivered by *p* in V' is also delivered by *q* in V'.

VSGMS9.1: *Sending View Delivery:* If a process *p* delivers message *m* in view *V*, and some process q (possibly p = q) sends m in view V', then V = V'.

VSGMS9.2[59]: *Same View Delivery:* If processes *p* and *q* both deliver message *m*, they deliver *m* in the same view.

[59]If the application chooses to use the asynchronous FIFO buffer instead of implementing a blocking callback.

Algorithm 6.10: Consensus-Based Uniform View-Synchronous Communication (part 1)

Implements:
UniformViewSynchronousCommunication, **instance** *uvs*.

Uses:
UniformConsensus (multiple instances);
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *uvs*, *Init* ⟩ **do**
$(vid, M) := (0, \Pi)$; // current view $V = (vid, M)$
correct := Π;
flushing := FALSE; *blocked* := FALSE; *wait* := FALSE;
pending := $\emptyset$;
delivered := $\emptyset$;
forall m **do** $ack[m] := \emptyset$;
seen := $[\perp]^N$;
trigger ⟨ *uvs*, *View* | (vid, M) ⟩;

upon event ⟨ *uvs*, *Broadcast* | m ⟩ **such that** *blocked* = FALSE **do**
pending := *pending* ∪ $\{(self, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, vid, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, id, s, m] ⟩ **do**
if $id = vid \wedge$ *blocked* = FALSE **then**
$ack[m] := ack[m] \cup \{p\}$;
if $(s, m) \notin$ *pending* **then**
pending := *pending* ∪ $\{(s, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, vid, s, m] ⟩;

upon exists $(s, m) \in$ *pending* such that $M \subseteq ack[m] \wedge m \notin$ *delivered* **do**
delivered := *delivered* ∪ $\{m\}$;
trigger ⟨ *uvs*, *Deliver* | s, m ⟩;

Fig. 33: Fig. 4: Consensus-based Uniform View-Synchronous Communication (part 1) [CGR11]

Figures 33 and 34 (Fig. 33, Fig. 34) outline the pseudo code for a uniform view-synchronous communication protocol presented in ([CGR11]) on which the current VSGMS implementation is based. The modified generic Consensus algorithm from (Group Communication System based on Consensus) is used to agree upon the membership set as well as the unstable messages of the previous view. A flush-phase where unstable messages are exchanged between members of the group through reliable broadcast[60] before Consensus is initiated reduces the probability that a coordinator may exclude unstable messages in its proposal and subsequent decision. This type of flushing should not be confused with the previously mentioned *flush*-protocol from ([BSS91]). If processes only forward their unstable messages to the coordinator of a Consensus round failures or wrongful suspicions can exclude estimates and consequently unstable messages may have to be discarded. The initial exchange of these unstable messages together with the coordinator's *ProposalChooser* function, that forms the union of all unstable messages found amongst the received estimates, can largely prevent this situation from occurring.

[60]Either in a single, large message or through FIFO-reliable broadcast followed by a “flush v” message akin to the *flush* protocol.

Algorithm 6.11: Consensus-Based Uniform View-Synchronous Communication (part 2)

upon event $\langle\ \mathcal{P}, \mathit{Crash} \mid p\ \rangle$ **do**
    $\mathit{correct} := \mathit{correct} \setminus \{p\}$;

upon $\mathit{correct} \subsetneq M \wedge \mathit{flushing} = \text{FALSE}$ **do**
    $\mathit{flushing} :=$ TRUE;
    trigger $\langle\ \mathit{uvs}, \mathit{Block}\ \rangle$;

upon event $\langle\ \mathit{uvs}, \mathit{BlockOk}\ \rangle$ **do**
    $\mathit{blocked} :=$ TRUE;
    trigger $\langle\ \mathit{beb}, \mathit{Broadcast} \mid [\text{PENDING}, \mathit{vid}, \mathit{pending}]\ \rangle$;

upon event $\langle\ \mathit{beb}, \mathit{Deliver} \mid p, [\text{PENDING}, \mathit{id}, \mathit{pd}]\ \rangle$ **such that** $\mathit{id} = \mathit{vid}$ **do**
    $\mathit{seen}[p] := \mathit{pd}$;

upon (forall $p \in \mathit{correct} : \mathit{seen}[p] \neq \perp) \wedge \mathit{wait} = \text{FALSE}$ **do**
    $\mathit{wait} :=$ TRUE;
    $\mathit{vid} := \mathit{vid} + 1$;
    Initialize a new instance $\mathit{uc.vid}$ of uniform consensus;
    trigger $\langle\ \mathit{uc.vid}, \mathit{Propose} \mid (\mathit{correct}, \mathit{seen})\ \rangle$;

upon event $\langle\ \mathit{uc.id}, \mathit{Decide} \mid (M', S)\ \rangle$ **such that** $\mathit{id} = \mathit{vid}$ **do** // install new view
    forall $p \in M'$ such that $S[p] \neq \perp$ **do**
        forall $(s, m) \in S[p]$ such that $m \notin \mathit{delivered}$ **do**
            $\mathit{delivered} := \mathit{delivered} \cup \{m\}$;
            trigger $\langle\ \mathit{uvs}, \mathit{Deliver} \mid s, m\ \rangle$;
    $\mathit{flushing} :=$ FALSE; $\mathit{blocked} :=$ FALSE; $\mathit{wait} :=$ FALSE;
    $\mathit{pending} := \emptyset$;
    forall m **do** $\mathit{ack}[m] := \emptyset$;
    $\mathit{seen} := [\perp]^N$;
    $M := M'$;
    trigger $\langle\ \mathit{uvs}, \mathit{View} \mid (\mathit{vid}, M)\ \rangle$;

Fig. 34: Consensus-Based Uniform View-Synchronous Communication (part 2) [CGR11]

3.5.4 Simulation

Simulations based on MINHA

The ability to simulate and test protocols and protocol stacks is an essential component of the prototype eGCS design. In the design proposal of the previous chapter (3.4) two different simulation approaches were discussed and a solution based on the MINHA framework (see 2.4.5) chosen. MINHA is a middleware testing platform that employs bytecode rewriting to transform regular Java code so it executes in a simulated environment. MINHA allows the creation of virtual hosts (each with their own virtual JVM) that are interconnected through a simulated network. It is therefore possible to define scenarios where a number of hosts instantiate a protocol composition and perform a set of predefined actions without requiring the modification or replacement of components in the stack with simulated counterparts. MINHA allows the scheduling of method invocations in simulation time on objects residing on the virtual hosts through special proxies. At the time of writing this thesis the MINHA framework itself does not provide dedicated *fault injection* mechanisms.

To experiment with the framework and evaluate its adaptability for the *e*GCS prototype it was modified to allow for the specification of specific and randomized packet loss as well as crashing a process (in this case the simulated JVM) through a method call. Figures 36 and 35 (Fig. 36,Fig. 35) contain code that relies on these modifications. *TestScenarios* itself are not dependent on MINHA and simply provide a basic structure for defining tests. Several basic test scenarios based on the MINHA framework were implemented with different goals in mind. Of the presented two examples the first (A GCS Test Scenario) outlines one of several tests that were used to simulate the prototype *e*GCS protocol stack during development. The primary goal was to verify that the developed composition is in fact compatible with the rewriting process and simulated processes produce the same or an acceptably similar[61] output in both the simulated and regular environment. The second example (A Rudimentary Fault Injection Experiment) shows a rudimentary test of how *fault injection* could be used in test scenarios. Finally an approach how invariants may be defined is also presented (Logging Events and Defining Invariants).

A GCS Test Scenario

The following code snippet (Fig. 35) shows a test scenario using MINHA where testing was done on the prototype GCS stack. The abstract class *MinhaTestScenario* was implemented as a basis for defining tests relying on the MINHA framework and contains boilerplate code. A concrete test scenario such as *GCSTest* needs to implement the abstract method *configureHosts* that takes the set of simulated *Hosts* as a parameter. *HostProgram* is an Interface tagged with the *Global*[62] *Annotation* and is used to create an *Entry. Entries* provide a proxy for objects residing on the simulated process so the simulation driver code can schedule method invocations from outside the simulation. The constraint placed upon such “Global” interfaces is that they only reference other classes defined as *Global*[63] in MINHA.

[61]Timing differences and other variations will cause discrepancies between the simulated and real execution. Acceptably similar refers to execution behavior that is within the expected range of behavior seen during regular execution.
[62]Classes marked with the *pt.minha.api.sim.Global Annotation* are ignored by the custom *ClassLoader* and excluded from the rewriting process.
[63]Primitive types are also permitted.

```
1       public class GCSTest extends MinhaTestScenario {
2
3               @Override
4               public void configureHosts(Collection<Host> hosts) {
5               if ( hosts == null) {
6                       // TODO
7                       throw new RuntimeException();
8               }
9               // create the same entry for all hosts
10              // and execute it at the start of the simulation
11              long startTime = 0;
12              for ( Host h : hosts){
13                      try {
14      Entry<HostProgram> e = h.createProcess().createEntry(
15                              HostProgram.class, GCSTestHostProgram.class.getName());
16              // execute at 0 sim time
17              // NOTE : multiple entries are queued and executed in their entry time order
18              // (single threaded) so two entries at 0 will be executed after each other,
19                      // the second starting when the first returns, and not at time 0!
20                              e.atNanos(startTime).queue().startHost(h,hosts);
21                              if ( random.nextInt(10) > 5 ){
22                                      h.crashCPU(((long)random.nextInt(1500000000))*10);
23                              }
24                              startTime+=1000000000; // start next host after 1 seconds
25                      } catch (Exception e){
26                              failed = true;
27                              e.printStackTrace();
28                      }
29              }
30      }
```

Fig. 35: Code snippet of GCSTest that was used to experiment with the GCS stack in a modified version of MINHA that allows for the simulated failure of processes

The reason for this requirement is to ensure simulation integrity so that rewritten objects are not accessed from outside the simulation sandbox. The other requirements are that the class implementing the interface, in this case *GCSTestHostProgram*, is not global (or else it is not part of the simulation) and provides a constructor taking no parameters. Once *GCSTestHostProgram* is started by invoking the method *startHost* through the Entry proxy a very basic program is executed that attempts to join a group, send some messages using view synchronous reliable FIFO ordered multicast and then leaves the group and shuts down the protocol stack. Some of the processes are crashed at a random time using the method *crashCPU* which was added to the MINHA API (Fig. 35 line 22). This simulation is intentionally kept simple as the goal was to test for incompatibilities and other issues when using the prototype *e*GCS protocol stack in combination with MINHA.

A Rudimentary Fault Injection Experiment

Figure 36 (Fig. 36) shows a code snippet from a simple test scenario called *DatagramTestScenario* that was used for testing how fault injection may be integrated into the simulation. Two simulated processes, one a sender that transmits UDP packets, the other a receiver that polls for incoming UDP packets and prints them to the console, are defined. *DatagramTestScenario* instantiates a new MINHA simulation *World* and uses the provided *HostWrapper* class to create two virtual hosts running the respective target programs. Through the *HostWrapper* it is possible to invoke methods of the (global) *SimProgram* interface at a specified simulation time. It is important to note that the time of invocation may be delayed if a previous invocation on the same object has not yet finished its computation. This is of course an expected behaviour as the virtual host is simulating the main thread of the process when it is executing these invocations and a backwards movement in time would cause an invalid simulation state. (MINHA reflects the actual execution time, so the execution of “real” code advances simulation time accordingly. A scheduled execution can hence only ever

commence at a point in time greater than or equal to that of the return time of a previous execution). The example then initializes the programs and also specifies a number of datagram packets that are to be dropped. Several “execute” method invocations are registered for the host containing the sender process after which the actual simulation run is started.

```
1       World world = new World();
2       // create hosts in the simulation that run the specified class
3       HostWrapper<SimProgram> host1 = HostWrapper.getHostWrapper(
4               SimProgram.class, SenderProcess.class, world, "host1","192.168.2.1");
5       HostWrapper<SimProgram> host2 = HostWrapper.getHostWrapper(
6               SimProgram.class, ReceiverProcess.class, world, "host2","192.168.2.2");
7       // invoke methods on the simulated hosts
8       // invocations are handled sequentially and if the target method blocks
9       // other invocations on that host may be delayed, possibly indefinitely
10      host1.invokeAt(0).initProcess(new String[] { "5000", "192.168.2.2", "5000" });
11      host2.invokeAt(0).initProcess(new String[] { "5000" });
12      // address used for specifying the socket at which packets should be dropped
13      InetSocketAddress s = InetSocketAddress.createUnresolved("192.168.2.1", 5000);
14      // drops the n+1th packet sent on the socket
15      // e.g. dropPacket 0 drops the first packet sent on the socket
16
17      // host1.getHost().dropPacket(s, 0);
18      // host1.getHost().dropPacket(s, 1);
19      // host1.getHost().dropPacket(s, 2);
20      host1.getHost().dropPacket(s, 3);
21      host1.getHost().dropPacket(s, 4);
22      host1.getHost().dropPacket(s, 5);
23      // host1.getHost().dropPacket(s, 6);
24      // invokes the method call at n nanoseconds simulation time
25      host1.invokeAt(1000).execute(new String[] { "hello", "world" });
26      host1.invokeAt(2000).execute(new String[] { "3" });
27      host1.invokeAt(3000).execute(new String[] { "4" });
28      host1.invokeAt(4000).execute(new String[] { "5" });
29      host1.invokeAt(5000).execute(new String[] { "6" });
30      host1.invokeAt(6000).execute(new String[] { "7" });
31      // run simulation
32      world.run();
```

Fig. 36: Code Snippet of DatagramTestScenario.java

Logging Events and Defining Invariants

In the design proposal it was also outlined that the definition of test scenarios with global and local invariants lies at the heart of the testing approach. A *TestScenario* executes several instances of the target protocol stack with a simulated application that submits relevant *Events* to a global event list to form a reconstructed global execution. *Invariants* are defined on the set of Events generated during a test run to test the correctness of the execution. After the simulation has completed the Invariants are evaluated and their outcome is used to determine if the scenario as a whole has passed or failed. To obtain meaningful information the simulated processes have to generate Events and add them to the global event list for further evaluation. This can be achieved through a callback that is safely added to the simulation via an *Exit* proxy. Callbacks must also be defined as *Global* as they are accessed from within the simulated environment. The following code example (Fig. 37) illustrates how both an Entry and Exit proxy are defined and the callback is passed to the simulated process through a method invocation. The callback can be used to record important events in the global event list. Classes extending Event need to adhere to the rule of only referring to other classes marked as Global or use primitive types. Once the simulation has completed Invariants are applied to the global event list to test if the execution was correct.

```
1       final Queue<Event> events = new ConcurrentLinkedQueue<Event>();
2               try {
3                       World world = new World();
4                       Host host = world.createHost();
5                       pt.minha.api.Process proc = host.createProcess();
6                       Entry<LoggingInvocation> e = proc.createEntry(
7                               LoggingInvocation.class, Program.class. GetName());
8                       // define an Exit proxy for the EventCallback
9                       Exit<EventCallback> x = proc.createExit(EventCallback.class,
10                              new EventCallback() {
11                                      @Override
12                                      public void registerEvent(Event e) {
13                                              events.add(e);
14                                      }
15                       }
16                      );
17                      // asynchronous invocation that passes the callback proxy
18                      // to the simulated process
19                      // the callback is set to report synchronously
20                      // so a consistent snapshot of the simulation state is observable
21                      // an asynchronous callback is produced with x.report()
22                      e.queue().myMethod(1, x.callback());
23
24                      world.runAll(e);
25                      e.getResult();
26                      world.run();
27                      world.close();
28                      // Create Invariant for the global execution
29                      Invariant a = new MyInvariant(events);
30                      // check if the Invariant holds
31                      failed = !a.call();
32              } catch (Exception e1) {
33                      e1.printStackTrace();
34              }
```

Fig. 37: Example of how Events could be logged using an Exit proxy and Invariants may be used.

Initial results using MINHA

The MINHA framework has proven itself to be an invaluable tool during the development of the *e*GCS prototype. Only little additional coding effort in the form of simulation driver code was required to be able to perform simulations of distributed processes running the targeted protocol stacks entirely in a discrete event simulation. Unfortunately several issues were also encountered that raise important questions if the framework is to be used in an educational context. Experiments with fault injection revealed that closing *Hosts* and *Processes* from the driver code to simulate process crashes can cause problems in combination with a *finally block* in a *try-catch* statement[64]. Some language elements like the use of *TimeUnit* may cause the rewriting process to fail entirely while others such as using custom serialization or relying on an *ExecutorService* may exhibit incorrect behaviour that is not immediately apparent. The following list outlines several of the issues discovered while experimenting with the MINHA framework.

1. Instantiating a *java.lang.Thread* with a *java.lang.Runnable* object in a certain way creates an Exception. { see https://github.com/jopereira/minha/issues/2 } (Resolved)
2. Custom serialization using *readObject/writeObject* methods will only work if the class directly implements *Serializable* and does not inherit the interface.
   { see https://github.com/jopereira/minha/issues/3 } (Open)

[64]Under certain conditions a simulator-specific exception is caught by the *finally block*, thereby escaping the simulation sandbox. See {https://github.com/jopereira/minha/issues} for more details.

3. Synchronizing on a Collection using the keyword *synchronized* produces a RuntimeException.
   { see https://github.com/jopereira/minha/issues/5 } (Resolved)
4. The method *tryLock* in java.util.concurrent.locks.ReentrantLock does not recognize it when the current Thread already holds the lock.
   { see https://github.com/jopereira/minha/issues/10 } (Resolved)
5. Deamon Threads are not handled properly by the simulation.
   { see https://github.com/jopereira/minha/issues/12 } (Resolved)
6. Closing a Host or Process in the simulation can cause a RuntimeException.
   { see https://github.com/jopereira/minha/issues/13 } (Resolved)
7. Finally blocks are unintentionally executed if a Host or Process is closed.
   { see https://github.com/jopereira/minha/issues/14 } (Open)
8. Using java.util.concurrent.TimeUnit causes a VerifyException during the rewriting process.
   { see https://github.com/jopereira/minha/issues/16 } (Open)
9. The behaviour of a *synchronized* block in combination with *wait* and *notify* methods of *java.lang.Object* is not as expected.
   { see https://github.com/jopereira/minha/issues/17 } (Resolved)

A fundamental problem in regards to these issues is that erroneous behaviour caused by MINHA may not be easily associated with the use of the framework. The debugging process is rendered more difficult because instrumented code that is executed no longer matches the one presented in the development environment. Such drawbacks are acceptable for the development and testing of group communication systems and other complex distributed middleware by more experienced programmers, however in an educational scenario these issues become problematic. MINHA is actively being developed and most of the reported issues that were encountered during the implementation of the prototype have already been resolved. Nevertheless for the *e*GCS prototype and the goals set out by this thesis the uncertainties in respect to potential bugs and issues students may encounter currently outweigh the advantages gained. Alternatively it is possible to restrict the freedom students have when implementing protocols by giving them clear guidelines which libraries and language functions are safe to use.

Alternative Simulation Approaches

In (3.4.5) two different simulation approaches were discussed as potential solutions for the *e*GCS prototype. Instead of using MINHA to provide an environment where regular code is automatically transformed through the framework so it is synchronized in respect to the (discrete event) simulation, this process can also be achieved manually. Protocol layers of the GCS stack need to be replaced by simulated or emulated counterparts to allow for a local simulation of a distributed execution. In the case of emulation the simulated components need to be real-time capable to avoid timing issues. As described in part one emulation is unproblematic until the case is reached where the amount of overhead incurred by running several instances locally as well as the simulated network causes the permissible time bounds to be exceeded. If only a moderate number of processes

are used for simulation purposes the hardware requirements for students should be low enough to make emulation a feasible approach.

On the other hand if protocol layers are simulated based on a discrete event simulation the execution of regular code needs to be synchronized in respect to this simulation to avoid timing issues and incorrect behaviour. This is especially important if students rely on time-out values or other forms of explicit timing. The provision of a library that wraps such functionality and can be switched between regular execution and simulations is one possible solution that is, for instance, found in the Neko ([USD01]) framework. Any timing assumptions must then be directed through this particular library.

An interesting approach may be the integration of a failure detector abstraction into an asynchronous lower level networking protocol simulation. A simulated communication layer with similar guarantees to those of the current group communication simulator (see 3.2.2) could then be used if it is ensured that the entire GCS protocol stack solely relies on the failure detector instead of using explicit timing assumptions. Switching between simulation and regular execution would merely require the replacement of the network protocol and failure detector component with real counterparts.

The chosen testing approach of using local and distributed invariants on a reconstructed global execution is not directly dependent on a particular simulation model as long as it produces valid and representative executions. It is preferable to have access to the global state of the system during the execution of tests as this simplifies the reconstruction of a consistent global execution that is later used for evaluating the defined invariants.

3.5.5 Testing Based on Global Executions and Invariants

The following classes make up the basic testing API:

- **TestScenario**

  TestScenario is an abstract class used as a basis for defining test scenarios. It is parametrized with the type of the to-be-tested instances and implements the *java.util.concurrent.Callable* interface with *GlobalExecution* as the generic type. The constructor accepts a *Collection* of objects matching the generic type parameter of the TestScenenario instance.

- **GlobalExecution**

  A *GlobalExecution* is returned whenever a test scenario has been executed. GlobalExecution provides the ability to register events which are ordered according to their creation time. It implements methods for returning both the global ordered list of events as well as ordered local event histories of different processes.

- **Event**

  Events are used to document relevant information for the GlobalExecution. During the instantiation of an Event its creation time is set using a monotonically increasing timer based on the system time.

- **Invariant**

  Finally the abstract class *Invariant* serves as a basis for defining invariants for generated global executions. It implements the Callable interface with the class *Boolean* as the generic type parameter. Invariants take a GlobalExecution and a boolean value of the expected result based on evaluating that GlobalExecution as constructor parameters.

As the previous subsection has outlined a simulation approach using MINHA may lead to issues that currently render its use in an educational setting problematic. A simulation approach based on replacing different layers in the stack with simulated counterparts is hence required. The following figure (Fig. 38) illustrates how a test scenario may interact with protocol stack instances. A simulated application (*TestingProcess*) that is controlled by the TestScenario is placed on top of the targeted protocol stacks. Apart from allowing external input to be made it is also responsible for logging all relevant events it perceives to the GlobalExecution contained within TestScenario. Furthermore specifically for the test scenario additional testing layers may be introduced into the stack. These layers can report back additional information or allow the TestScenario instance to exert control over them. An example for this could be a protocol layer that buffers sent and received messages instead of forwarding them, allowing the TestScenario to manually control the set and sequence of messages that are being passed on.

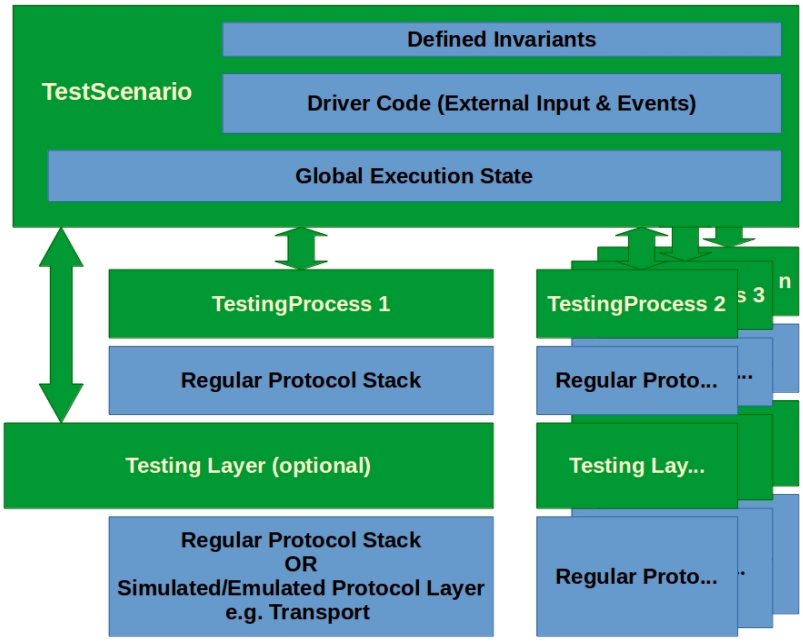

Fig. 38: Interactions between Test Scenario and Protocol Stack Instances

The following example outlines how a test scenario using student developed protocols may be realized. Students are tasked to extend or implement a factory class that creates GCS stack instances containing their self-developed protocols. This factory is then used by the testing code to instantiate different TestScenarios with the necessary protocol stack instances required for conducting the test. Either predetermined TestScenarios are used or students may have to create their own

implementations where external input such as the sending of messages or the failure of processes (if the API permits it) is specified. When timing this external input TestScenarios can rely on their global knowledge of all stack instances and their states to orchestrate desirable testing situations. Important Events from each instance are logged to a GlobalExecution, which keeps a time-ordered list of Events based on a shared global clock. Once a TestScenario completes it returns the recorded GlobalExecution for that test so that Invariants may be applied to it. The results of evaluating the different invariants are finally used to determine if the used protocol stack behaved according to its specification.

As part of the *e*GCS prototype development process and as a use case the original ADS laboratory simulator's modality with static group semantics was implemented using the header-driven composition model and parts of the *e*GCS protocol stack together with this described testing approach. A more detailed description can be found in section (3.5.7 Emulating the Original ADS Laboratory Simulator as a Use Case).

3.5.6 Code Structure of the *e*GCS Prototype

Important Class Packages

- *framework.composition.headerbased*

  This package contains all the relevant components for the header-driven protocol composition model. Base classes for the HeaderHandler, Handlers and their corresponding Headers, header-based messages as well as a basic protocol template are provided.

- *framework.consensus*

  Contains a generalized Consensus API as well as an implementation of the modified Chandra-Toueg Consensus discussed in (3.5.3). The protocol relies on a *ConsusProvider* to return Consensus instances for a particular *ConsensusID*. ConsensusIDs are made up of the (static) group participating in the consensus instance, a unique id, the dynamic type of the proposal, and a ProposalChooser function that is applied to that dynamic type for when the coordinator chooses an estimate as its proposal.

- *framework.failuredetector*

  Defines the basic API for failure detectors as well as different failure detector implementations and failure monitors used to update local unreliable failure detector modules. A protocol is also provided that transforms any given failure detector exhibiting weak completeness into one that exhibits strong completeness based on the transformation algorithm presented in ([CT96]).

- *framework.gms*

  Here the API for both static groups and dynamic group membership, including virtually synchronous communication, is defined and the package also contains the prototypical implementation of the VSGMS (see 3.5.3). An *AgreedGroup* interface serves as a base representation for static groups and views[65].

- *framework.network*

  This package provides an abstraction for the networking layer so that the protocol stack can use a uniform API for sending and receiving messages over the network. Both a simulated network (requires that all protocol instances reside in the same JVM) and an implementation using network sockets (TCP) are included. It also includes the API for resolving logical addresses specified by processes and two implementations, one simulating address resolution while the other uses basic IP multicast.

- *framework.protocol*

  Contains various protocol implementations including asynchronous static group multicast, view synchronous multicast, basic multicast protocols and other implementations such as a ping protocol.

- *framework.testing*

  Defines the testing API that relies on invariants being applied to reconstructed global executions in order to verify correctness. TestScenarios are *callable*[66] objects that return a GlobalExecution which is a time-ordered list of events that occurred at individual instances during the test's execution using a global system time. Invariants are also callable returning a boolean value and take a GlobalExecution as their constructor parameter. The package *framework.testing.adslu* implements this testing approach while following the scope of the original ADS laboratory assignment, that is, having students implement various static group multicast protocols with different guarantees such as reliable, FIFO, and causal ordering and also test their implementations for correctness with self-defined tests.

Header-Driven Composition Class Structure

The following class diagram (Fig. 39) outlines the relevant classes for using the header-driven approach in the *e*GCS prototype. A HeaderHandler serves as the basis for developing protocol stacks and holds references to all Handler instances. It also performs the task of message dispatching where the next header is popped from the message's header stack and, based on the unique name, dispatched to the correct Handler instance. *HeaderBased* is an interface with basic functionality related to the header-driven approach that protocols should, but do not have to implement. It is often used as a base interface when specifying specific protocol API interfaces. The only requirement for implementing a protocol in the header-driven approach is the implementation of a Handler and the ability to pass on messages to the remote peers' HeaderHandlers. The latter can be satisfied by holding a reference to a lower level protocol which in turn may pass on the message until eventually

[65] It is only through a sequence of views that group membership changes, while each individual view represents a static group agreed upon by all members of the view (for *group membership* as it is understood in this thesis), hence the definition of AgreedGroup.
[66] Implementing the *java.util.concurrent.Callable* interface.

a protocol is reached that is able to perform a remote message dispatch. The upwards flow of messages does not have to be explicitly defined as the header-driven approach implicitly handles message delivery to the correct protocol based on the header stack of the message. Headers are not directly instantiated, instead being created by calling the method *invoke* defined in the Handler together with an object of the generic parameter type.

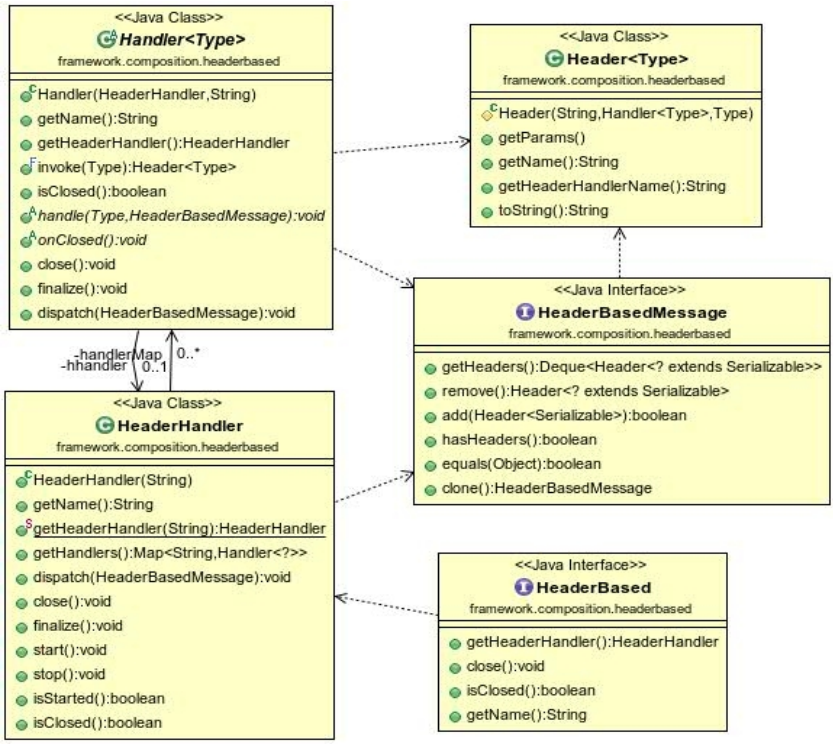

Fig. 39: Classes related to the header-driven Composition Model

Test Scenario Class Structure

Figure 40 (Fig. 40) shows classes related to the testing API. Some of the method signatures have been removed to provide a better overview. TestScenario serves as the basis for defining tests and *TestScenarioTemplate* is used specifically for the ADS laboratory use case described in (3.5.6). TestScenario implements the java.util.concurrent.Callable interface with the class GlobalExecution as a generic parameter. GlobalExecutions are used to record Events which are ordered based on a global system time. Events are registered with the global execution by the TestScenarioProcess whenever it sends or receives messages or when a simulated failure is caused by calling the *crash* method. Finally Invariant is used for the creation of invariant definitions that are applied to the GlobalExecution provided in its constructor.

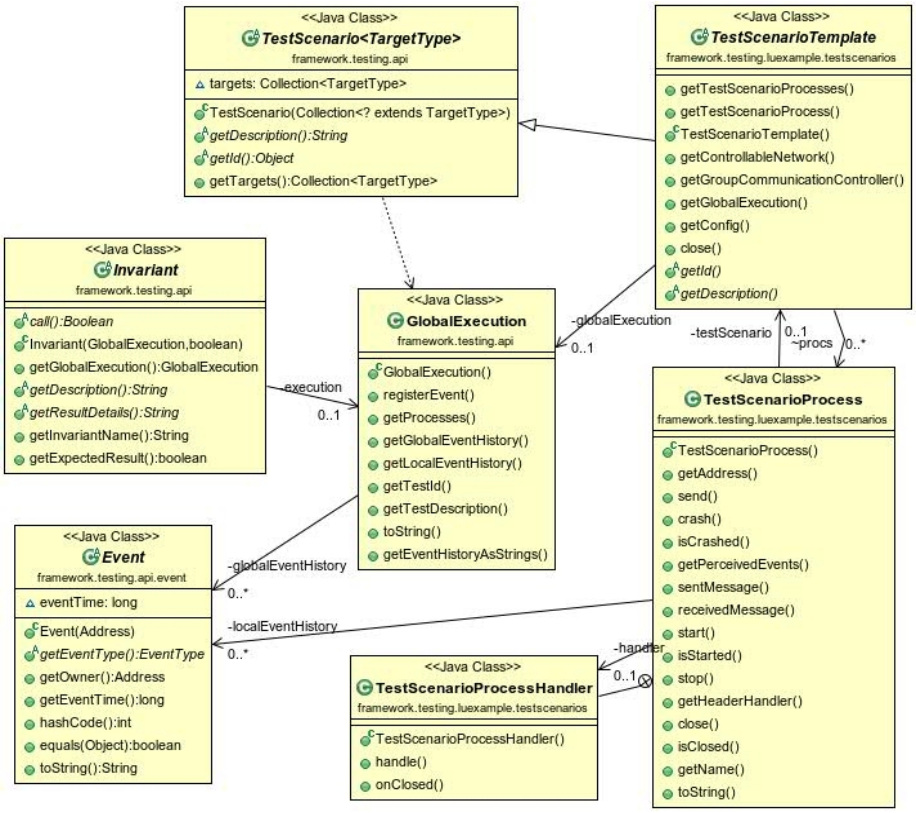

Fig. 40: Classes Related to the Testing API

Consensus Related Classes

Figure 41 (Fig. 41) contains classes and interfaces relevant to Consensus. A generically typed *Consensus* interface represents an instance of Consensus. The methods *propose* and *getDecision* return Objects of the type *java.util.concurrent.Future* as they represent asynchronous computations. Instances of Consensus are created using a *ConsensusProvider* which requires a ConsensusID. The ConsensusID wraps relevant information required to uniquely identify but also define instances of Consensus. It is generically typed with the same type as the to-be-created Consensus instance and contains an AgreedGroup specifying the participants, the class of the generic type for which the instance is to be parametrized, a unique id (implementing java.io.Serializable) and a ProposalChooser which is also parametrized with the same generic type. The ProposalChooser is part of the modified Chandra-Toueg Consensus described in section (3.5.3 Group Communication System) and under certain circumstances can allow the coordinator of a round of Chandra-Toueg Consensus to apply a function to the set of received estimates in order to generate a proposal rather than choosing one of the estimates. Some of the dependencies of such a modified Chandra-Toueg Consensus implementation (*HeaderBasedCTConsensus)* are shown including the FailureDetector API and a multicast protocol (*HeaderBasedMulticastProtocol)*. The latter has to be an implementation providing reliable multicast while the former needs to be at least a $\diamond\mathcal{S}$ failure detector.

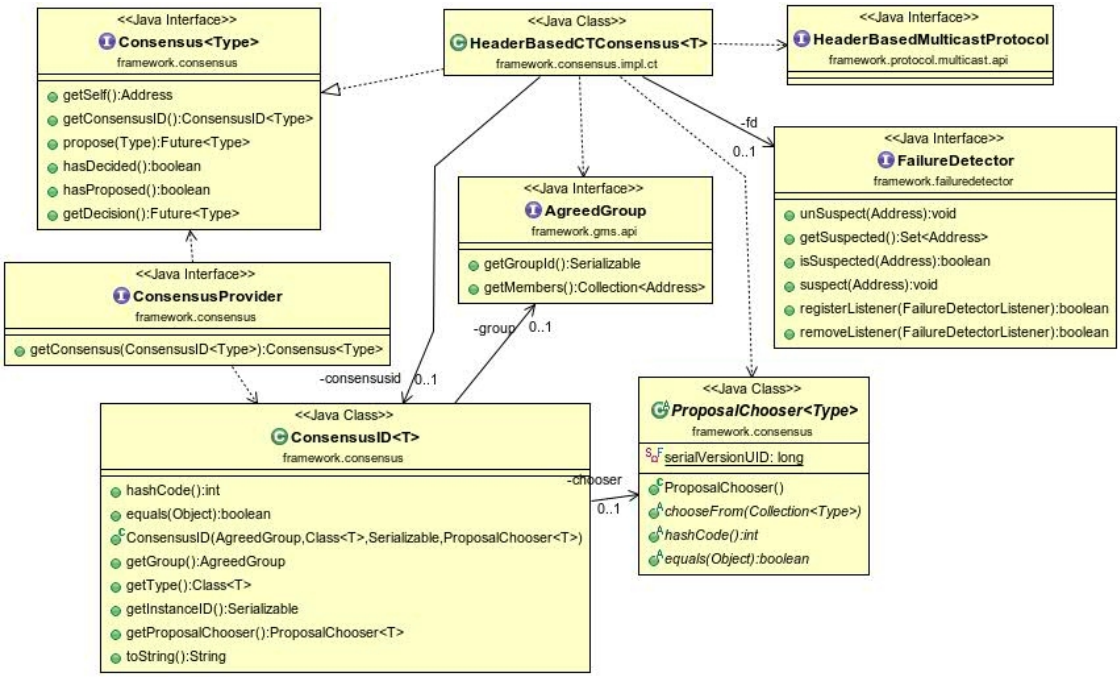

Fig. 41: Classes Related to Consensus

Group Communication Related Classes

In figure 42 (Fig. 42) important classes and their relationships in regards to group communication are shown. Details were sometimes left out to provide a better overview. AgreedGroup serves as a basic interface for describing static groups. The name AgreedGroup was chosen because the View interface is derived from it. Views in themselves[67] are an agreed set of members that does not change and static groups also form a predefined agreed upon set of members, hence it is logical to use the term *agreed group* as a common descriptor. *GroupCommunication* is an interface to indicate that a protocol provides group communication primitives. It defines the methods *groupSend* and *getAgreedGroup*, but also inherits other interfaces related to the header-driven approach such as the HeaderBased interface. *ViewSynchronousCommunication* extends GroupCommunication and is, as the name suggests, used for protocols implementing view-synchronous communication. It extends the *Blocking* interface which allows *BlockListeners* to be notified whenever they need to block and unblock. The *GroupMembershipService* interface is used as a basic API for defining group membership services. *VSGMS* is an implementation of the in section (3.5.3) described approach that combines both a group membership service and a view-synchronous communication protocol and is based on modified Chandra-Toueg Consensus.

[67]In the context of group communication as it is considered in this thesis new views may be installed or exist concurrently but the membership set of an individual view does not change.

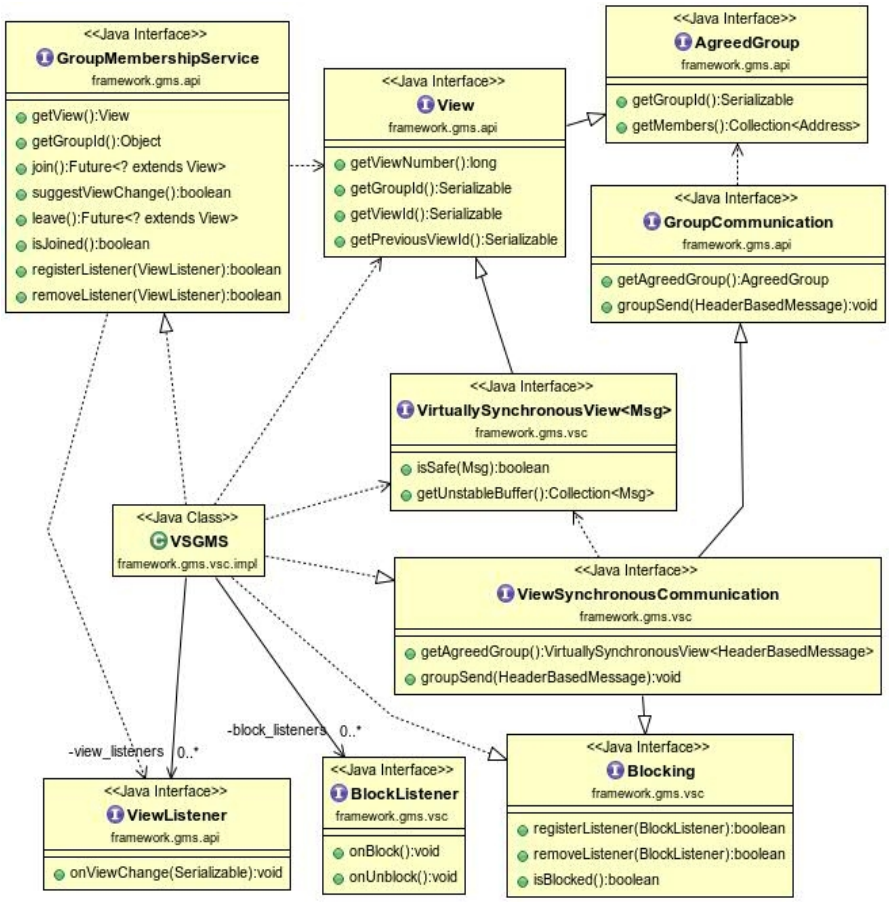

Fig. 42: Important Classes related to Group Communication

Classes Related to the ADS Laboratory Static Group Use Case

Finally, figure 43 (Fig. 43) shows important classes for the advanced distributed systems laboratory use case presented in the next section (3.5.7). Method signatures and other details were left out to keep the image overseeable. *ProtocolTester* is the main application provided to students for testing their protocol implementations. It relies on the *LabBaseGroupCommunicationProvider* for instantiating basic protocol stacks that are handed over to an implementation of *GroupCommunicationProvider*. The GroupCommunicationProvider returns a protocol stack of type *GroupCommunication* which is used by TestScenarioProcess to send and receive messages within TestScenarios. The *TestScenarioTemplate* serves as a basis for defining TestScenarios and is also shown in (Fig. 40). Both *ControllableNetwork* and the *GroupCommunicationController* can be used during a test to control special testing protocol layers that are included in the base protocol stack provided to students. ControllableNetwork allows *LinkProperties* to be specified which alter the behaviour of message transmission characteristics at the transport layer. *GroupCommunicationController* provides the ability to control the send and receive buffers of *ControllableGroupCommunication* protocol layer instances. This implementation requires that all stack instances are executed within the same JVM, which is satisfied by the current testing approach using ProtocolTester. Implementations of TestScenarioTemplate return a GlobalExecution which is evaluated using student defined Invariants.

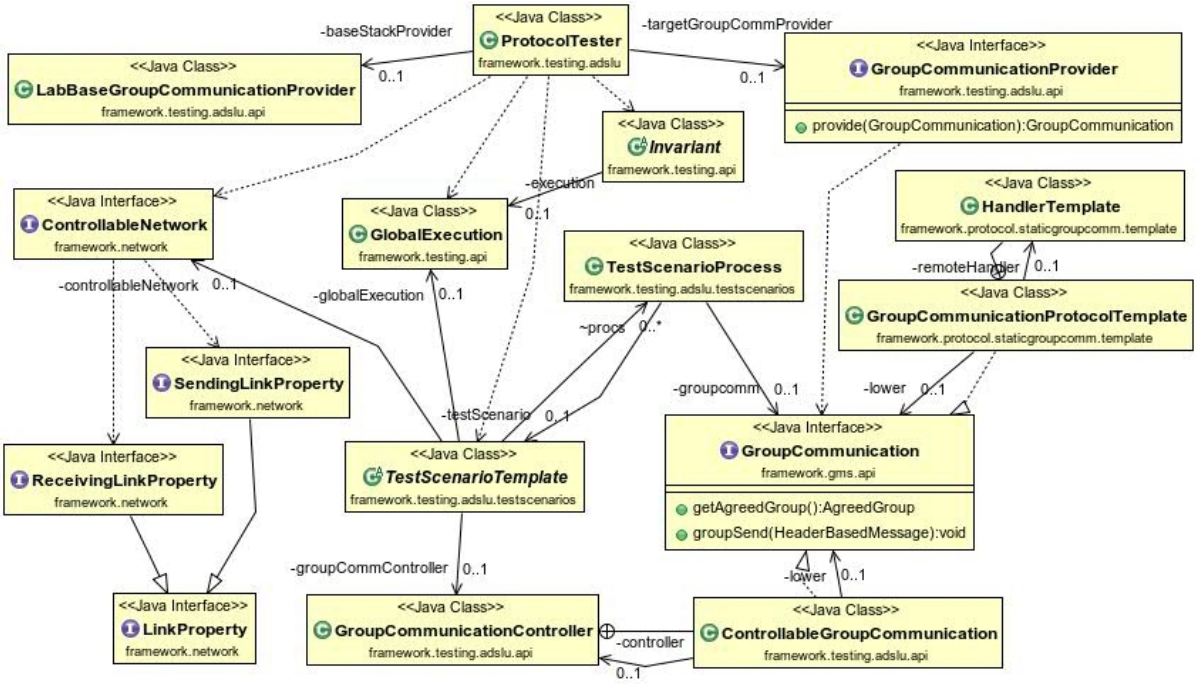

Fig. 43: Classes Related to the ADS Laboratory Use Case

3.5.7 Emulating the Original ADS Laboratory Simulator as a Use Case

Overview

An important part of the development process of the *e*GCS was to show that the modality of the original advanced distributed systems laboratory (reliable asynchronous multicast communication using a static group model) could be implemented with the proposed approach while also satisfying the set out goal of allowing student protocols to be used in realistic scenarios. For this task only a subset of the existing *e*GCS protocol stack is needed, however the proposed testing approach has to be concretised to support automatic and reproducible test scenarios in order to evaluate the correctness of student protocols. The previous section (3.5.6) includes a class diagram of relevant classes (Fig. 43) for this specific usage scenario.

Students are tasked to implement provider classes that add their own reliable multicast protocols on top of a preconfigured protocol stack which is provided to them. Through the header-driven approach lower level protocols are easily encapsulated so to students the entire stack is only visible as an instance of the type GroupCommunication. A basic protocol template for group communication protocols is included that takes a GroupCommunication instance as a constructor parameter. Once students have implemented the required protocols, namely non-uniform, uniform, FIFO, and causal reliable multicast, and added them to the corresponding providers the result can be tested using the provided testing framework or used in real applications. As a basic real application example a crude group chat was implemented which is briefly described at the end of this section. The testing approach is covered in detail in the next subsection.

Testing Approach for the Static Group Use Case

As outlined testing is based on a similar approach to that presented in ([FKK+05]) and follows the general setup presented in section (3.5.5). Student defined test scenarios, which are derived from a TestScenarioTemplate, add a simulated application layer (TestScenarioProcess) on top of unstarted protocol stack instances that are created using a specified GroupCommunicationProvider. To simplify the process of creating a consistent global execution all stack instances are created within the same JVM using a single test scenario instance, so that a shared global object may be trivially realized. Furthermore the global knowledge over all stack instances can be used by the test scenario to precisely orchestrate relative timings of external inputs based on the current global state. The simulated application layer is used to define external inputs (sending of messages and simulated failure of the instance) and log the locally perceived events (send and receive of messages and simulated failure) to the global execution contained within the test scenario. After a test scenario has been executed the global execution is returned and evaluated using student defined invariants.

Two important additional protocol layers are introduced for testing purposes, namely *ControllableGroupCommunication* and *ControllableNetwork*. In the header-driven approach lower level protocols are generally encapsulated by a protocol instance unless it exposes these references through its API. To allow access to both of these protocol layers their references are passed on from the point of composition[68] to the TestScenarioTemplate.

ControllableGroupCommunication is situated directly above the unreliable static group multicast stack and enables full control of send and receive buffers at every protocol instance. Relying on a shared JVM this protocol allows users to precisely manipulate the sending and receiving of messages within a test scenario in order to create reproducible test conditions. Messages can either be passed along the regular lower level stack or directly inserted into the receive buffer at the destination. The latter avoids any necessary timing assumptions on the underlying implementation. In the employed asynchronous communication system model where protocols are entirely driven by the sending and receiving of messages deterministic test scenarios can therefore be created using this method. The following figure (Fig. 44) shows where the ControllableGroupCommunication protocol layer is situated in the protocol stack. Arrows illustrate how messages travel from one stack to another. Controlled send means that the controller provides the ability to decide when to pass on messages to lower layers while controlled receive is used when delivering messages to upper layers. Simulated send and receive means that instead of passing a message on to lower layers it is directly delivered at the peer layer in the remote stack, foregoing the entire lower layers and transmission process.

[68] In this case the provider used to instantiate the basic protocol stack upon which students base their own implementations.

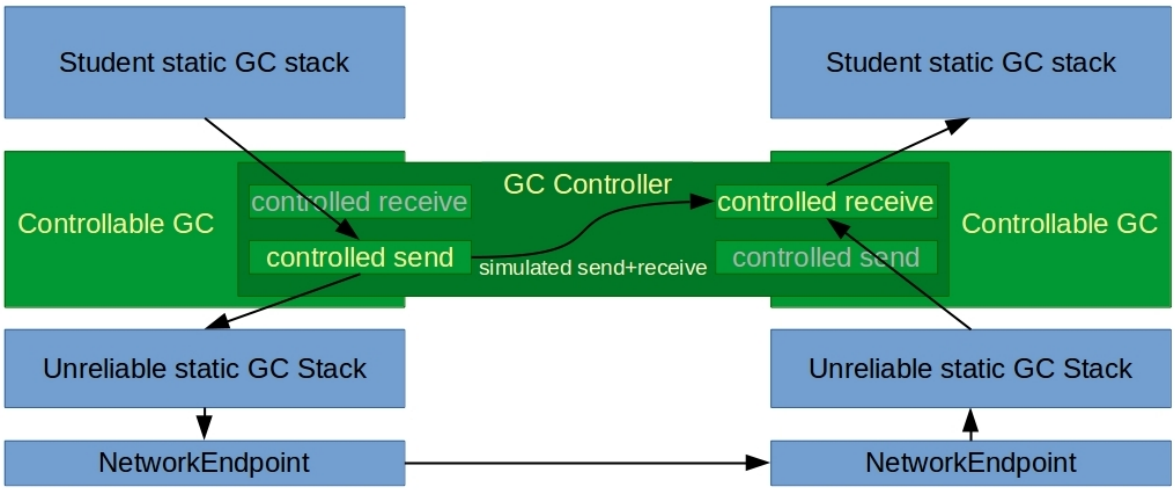

Fig. 44: Location of ControllableGroupCommunication in the testing protocol stack

ControllableNetwork provides the ability to manipulate messages right before they are are sent and after they are received on the transport layer through specifying *LinkProperties*. This is achieved by wrapping the *NetworkProvider* responsible for returning *NetworkEndpoints*, which serve as an abstraction for different concrete link implementations such as using TCP sockets or locally simulated links. *LinkProperties* are defined for the sending and receiving side of a directional link and are generally used to influence network transmission properties or for fault injection such as randomly dropping messages. ControllableNetwork is not dependent on a shared JVM.

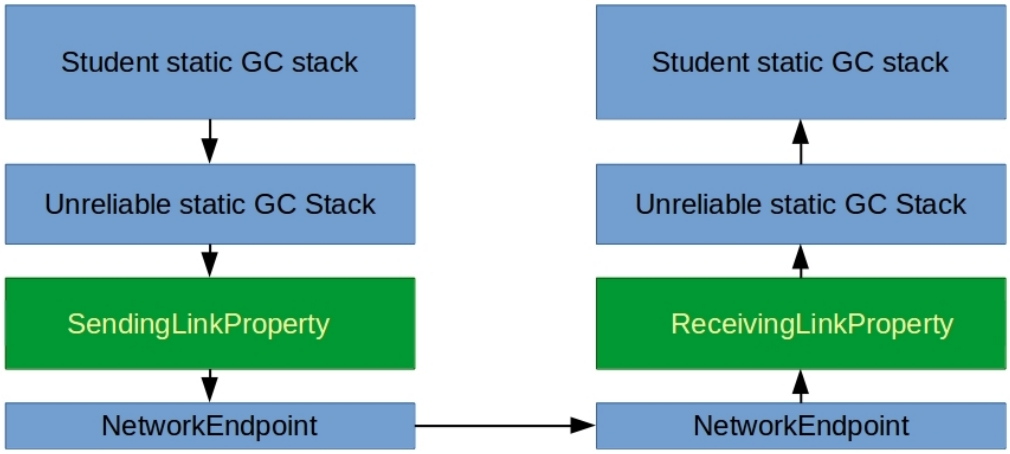

Fig. 45: Example where ControllableNetwork and LinkProperties are located in the testing protocol stack

Students are to use a specific ProtocolTester class for testing purposes which automates the process of executing several test scenarios. Through configuration files the relevant protocol providers for creating the group communication stacks, different test scenarios and their configurations, as well as the invariants that are to be applied to the reconstructed global executions from test scenarios are specified.

On the educator's side the correctness of student protocols is verified through a similar testing approach. Student protocols are evaluated using reference test scenarios and invariants while their self-defined test scenarios and invariants are tested against both correct and faulty implementations.

Figure (Fig. 46) provides an overview of the described testing architecture for the static group use case. Blue colour indicates regular protocol layers while green indicates layers specifically used for testing purposes. Test scenarios have the ability to exert control over these special testing protocol layers during the test's execution in order to be able to orchestrate desirable conditions and reproducible test runs. TestScenarios produce GlobalExecutions which are then evaluated using Invariants.

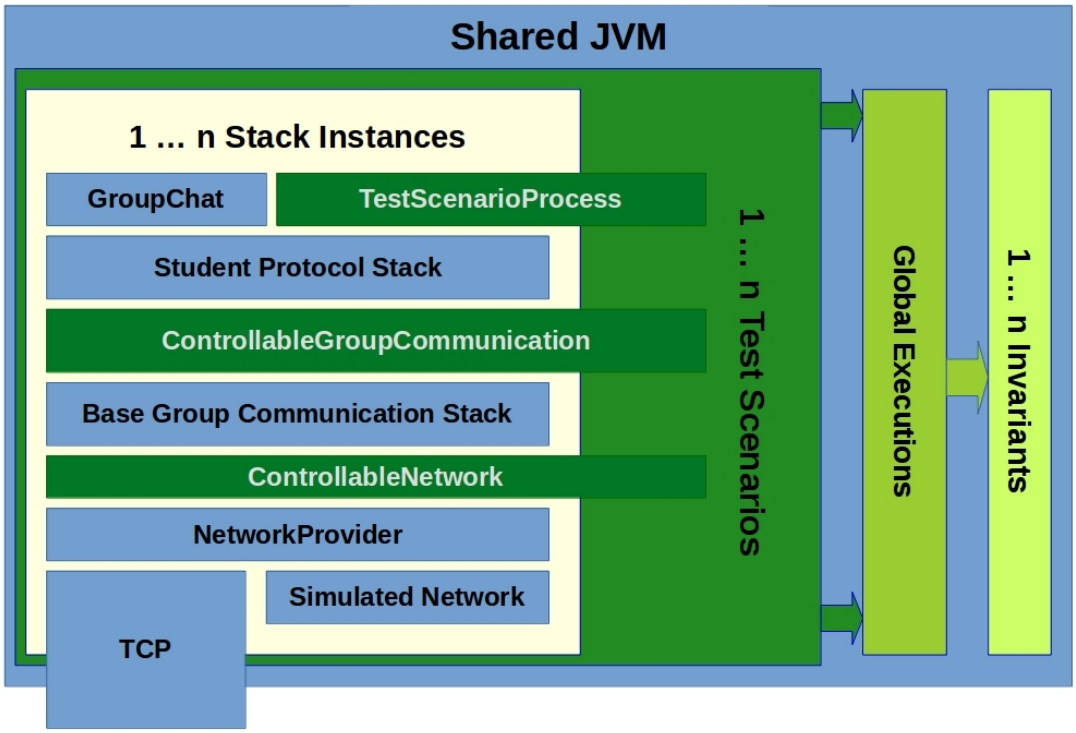

Fig. 46: Overview of the Testing Architecture for the Static Group Use Case

Real Application Example

One important goal of this thesis is to allow students to use their self-developed protocols in realistic scenarios. For the ADS laboratory use case a basic group chat application is used to visually highlight the effects of different delivery guarantees such as FIFO or causal order reliable multicast. The student defined protocol stack that is also used for testing can be directly employed for this example. Only the NetworkProvider and address resolution implementations need to be switched which is done in the configuration file. Figure 46 (Fig. 46) from the previous subsection also shows the regular protocol stack where the green protocol layers (namely ControllableNetwork, ControllableGroupCommunication and TestScenarioProcess) are omitted from the composition.

However if the ControllableNetwork layer is kept in the composition LinkProperties can be specified to simulate different communication behaviours. To illustrate this feature the chat application has the ability to load a predefined configuration for ControllableNetwork in order to modify the behaviour of the used communication links. Furthermore if all application instances and the underlying protocol stacks are started within the same JVM the ControllableGroupCommunication protocol layer could also be employed. More advanced testing scenarios are conceivable where both ControllableNetwork and ControllableGroupCommunication are combined with a GUI to allow for real-time manipulation and fault injection at the protocol stacks while being able to directly observe the effects on the application instances.

The following illustration (Fig. 47) shows an execution of the group chat application where all communication links to processes 2 and 4 (the bottom windows in the illustration) were modified to exhibit a large range of message delay (between 5 and 15 seconds added to each transmitted message). The effect on a FIFO ordered reliable static group multicast can be seen in the right lower window where the answer to process 1's message is received before the question has arrived. A causally ordered multicast would have delivered these messages in the correct order.

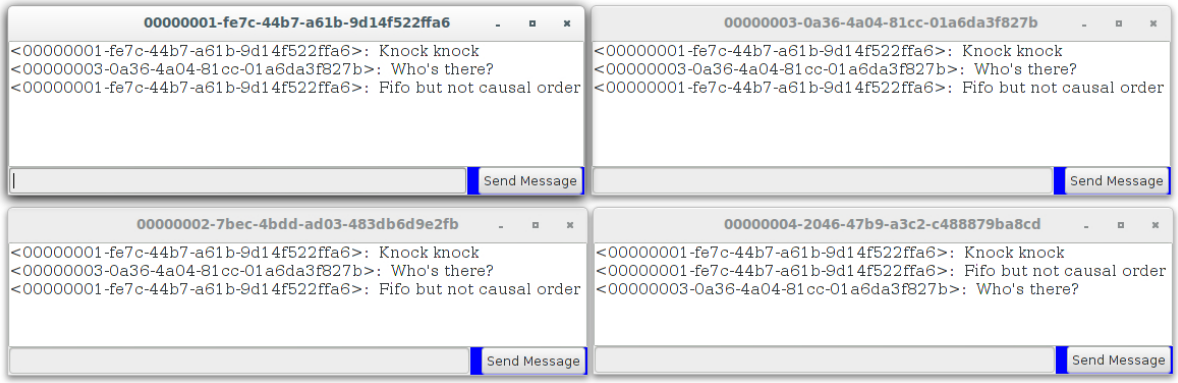

Fig. 47: Effects of FIFO Ordered Multicast in the Presence of Varied Message Transmission Times

3.5.8 Development Status of the *e*GCS Prototype

During the development phase the *e*GCS prototype has undergone two iterations. A first version was created that implements the entire envisioned *e*GCS protocol stack including a fully functional VSGMS layer and view synchronous protocols relying on it. This initial prototype however does not focus on ease of use and employs a less strict interpretation of the header-driven protocol composition approach. It was used to test the general feasibility of the design proposal and provides a template for the final version. The following example shows the group chat application from the previous section (3.5.7) using this first *e*GCS protocol stack. Because of the primary component group membership model[69] that is used the application will block if half or more of the processes fail within the same view. In scenarios where only a few instances are part of the group this effect is of course immediately apparent, but also to be expected. Other applications such as the JGroups (see 2.3.2) demo *org.jgroups.demos.Draw* were also modified to use the protocol stack when experimenting with the prototype.

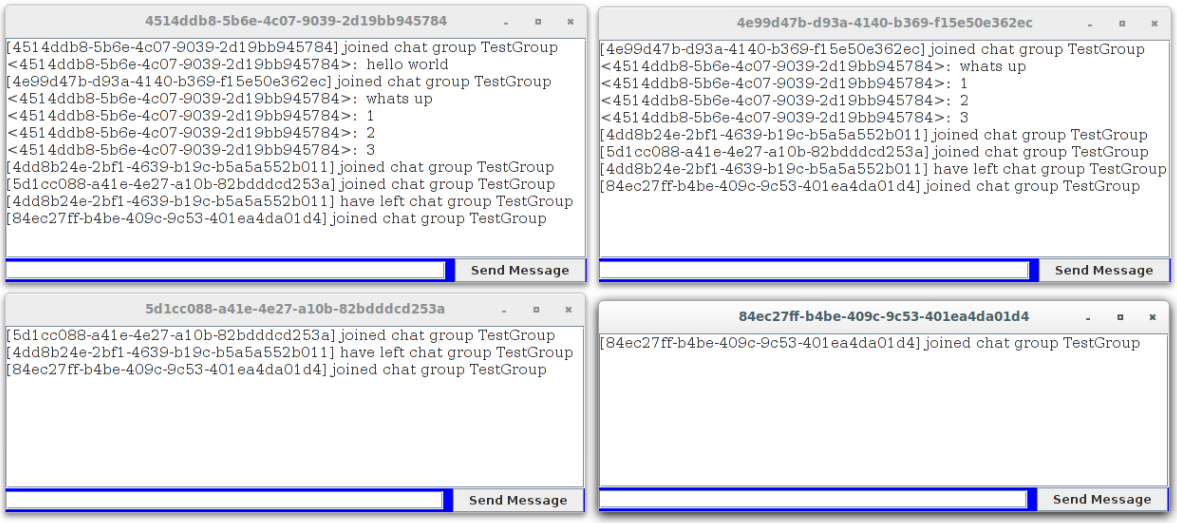

Fig. 48: Group Chat Application using the initial *e*GCS Protocol Stack

In a second iteration the composition model was reworked while also aiming for a simplification of the API to make the entire framework more accessible to students. Aspects such as relying on the dependency injection framework PicoContainer ([Pic11]) proved to introduce unnecessary complexity. Currently all protocol layers up to and including the modified Chandra-Toueg Consensus are finalized while the VSGMS layer is still in a prototypical state. The ADS laboratory use case relies on this second version of the *e*GCS protocol stack for its functionality.

[69] Specifically the Consensus instance trying to reach agreement on the next view and the set of messages seen will block.

3.6 Outcome

Part two of this thesis has covered the development of an *e*GCS prototype aimed at extending a group communication simulator for an advanced distributed systems laboratory. The prototype was implemented using a self-developed protocol composition framework based on the header-driven approach presented in ([BMN05]). The design employs a modified implementation of Chandra-Toueg Consensus to provide a combined protocol providing both group membership and view synchronous multicast communication. At the time of writing this thesis the chosen approach for simulating distributed executions using the MINHA framework is currently not ideally suited for an educational context. An alternative approach for performing simulations is therefore needed. The first version of the prototype *e*GCS provides a functional GCS protocol stack that can be used to implement reliable distributed applications. In the second iteration both the API and header-driven composition model were refined, however the VSGMS is currently still a prototypical implementation.

As a use case scenario and also to outline the general feasibility of the design the original setup of the advanced distributed systems laboratory, namely having students implement reliable static group multicast protocols in an asynchronous communication model, was realized using parts of the (second iteration) *e*GCS protocol stack and the proposed testing approach. Local simulations are conducted by replacing the lower level transport layer with a simulated counterpart. In contrast to the original simulator students are now able to use their self-developed protocols in realistic scenarios and can even draw upon basic fault injection mechanisms during real executions for further evaluation and testing.

No optimizations were made to the code to avoid introducing bugs, which is heavily impacting the system's performance and will need to be addressed in future iterations of the design. A detailed and more formal specification of the system is also outstanding and has to be made once the general design is finalized. As outlined it was an essential goal to develop a working prototype to test the feasibility of creating a composition framework and GCS implementation based on the header-driven approach. The results in this respect are promising as the creation of protocol stacks is straightforward and the upwards flow of messages is implicitly handled through the chosen design, reducing the potential for faulty configurations. Compared to the previous group communication simulator the complexity of implementing a protocol with the *e*GCS prototype is comparable, especially if a protocol template is provided. Of course practical experience and data on the educational effectiveness of this approach over other composition models is still required to be able to draw any conclusions.

In regards to the modified Consensus protocol used as a basis for the VSGMS protocol layer a next step would involve the extension of the current API and implementation to conform to all of the design aspects of the *general agreement framework* presented in ([HMRT99]). The provision of such a framework can help to simplify the implementation of protocols that are variants of the *agreement problem* and provides a basic building block for the development of reliable distributed applications.

Developing a GCS is a time-consuming and error-prone process. Within the time frame of this thesis it was only possible to implement a basic prototype of the envisioned *e*GCS. It has been shown that the header-driven approach is a suitable method for developing protocol stacks. More thorough investigations and experimentations in regards to the MINHA framework would be required to be able to confidently claim that an application in an educational context will not cause problems. Specifically it needs to be ensured that the vast majority of Java language functionality and libraries will not cause issues and behave as expected when using the simulation environment. In light of the difficulties with the chosen simulation approach alternative design options such as relying on Kompics or Neko for switching between simulation an real execution may also have to be revisited.

4. Conclusion

4.1 Results

The goal of this thesis was the extension of an existing group communication simulator for an advanced distributed systems laboratory. Initial research however made it clear that at the time of writing no directly related literature was available to inform the development of such an educational tool for the topic of group communication.

Part one (2 Educational Software Tools for the Topic of Group Communication) of this thesis therefore gives an overview of related works for the more general field of distributed computing and attempts to formulate initial guidelines for developing such tools. In particular educational simulation environments for distributed algorithms and protocols are considered. Preceding this overview is a detailed outline of the fundamental concepts behind group communication, relevant group communication systems and protocol composition frameworks and how simulation and testing of GCSs can be conducted. It has to be stressed that it was not a goal of this thesis to perform a survey of the different educational approaches found in relevant literature or conduct a meta-analysis on their effectiveness. Only informal observations on the covered works are made and their general designs compared and discussed. From these observations a*n interactive learning environment*, *visualization* and *realistic problem scenarios* are identified as often encountered educational design decisions. Furthermore the following requirements for an *e*GCS are proposed: 1) the provision of dynamic group semantics, 2) modular composition and an easy integration of new protocols, 3) A simple and straightforward API, 4) emulation or simulation of a distributed execution locally that exhibits similar characteristics to regular executions. Part one is concluded by outlining that none of the covered GCSs or protocol composition frameworks satisfy all the defined requirements while also providing a fully functional GCS stack.

Part two (3 Development of a Prototype Educational Group Communication System) covers the development of an *e*GCS prototype for a specific laboratory assignment. The design is informed by the findings of part one and various potential approaches are discusses in light of these findings. In particular the decision is made to use a self developed composition framework based on the header-driven approach and the Dependency Injection design pattern rather than to extend an existing GCS or protocol composition framework. Initial results with the prototype implementation suggest that the header-driven approach is a feasible composition model that allows for a straightforward development and composition of protocol stacks. The *e*GCS prototype's requirement of supporting dynamic groups and virtual synchrony can be satisfied through a single protocol combining the group membership service and view synchronous multicast communication. It is based on a

modified variant of Chandra-Toueg Consensus and provides primary component group membership. The initial prototype design relies on the distributed middleware simulation environment MINHA for simulation and testing in combination with the definition of global and local invariants based on a reconstructed global execution for verifying correct behaviour. MINHA was chosen as a simulation approach as it can avoid synchronization issues between regular code and simulated protocol layers and requires no modifications to the protocol stack if it is to be executed in the simulation environment. During the implementation phase it was however discovered that MINHA can produce hard to debug issues under certain circumstances, which currently makes it not ideally suited for an educational context.

While the results of this work present a prototypical implementation of an *e*GCS, the envisioned solution for simulating and testing GCS protocol stacks is not suited for an application in an educational context at this point in time. Alternative approaches towards providing simulation capabilities have been outlined, however without further modifications to the implementation the entire set out goal is currently not achieved.

4.2 Discussion

This thesis has made several important advancements towards solving the set out goals. Firstly, an overview of relevant literature on the design of educational tools for the topic of group communication is provided that can serve as a basis for future investigations. It can be considered an initial step towards forming a core literature for the topic area and clearly outlines the need for further research in this field. Formal classifications and comparisons of these educational tools as well as meta-studies on the educational effectiveness of different approaches and their governing pedagogy are required if a meaningful discussion is to be had. Until such work is conducted it is difficult to make informed design decisions for the development of educational tools for advanced distributed computing topics. The proposed design considerations and requirements for educational group communication systems are therefore very generalized and should only be seen as a rough guideline.

Secondly, a concrete design proposal and prototypical implementation of an *e*GCS is presented to solve the initial problem of extending the current group communication simulator. It is shown that the header-driven composition framework is suitable for developing GCS protocol stacks and may help to reduce the complexity of more traditional event-based composition models. The design of the GCS makes the implementation of reliable multicast protocols straightforward and offers enough flexibility for future modifications and adaptations if other coursework or course designs need to be realized. Currently a formal specification of the *e*GCS is still outstanding and will have to be made once the prototype design is finalized. While the envisioned simulation approach is currently not ideally suited for an educational context, the ongoing development and improvement of the MINHA framework might allow its use in the near future. Strict programming guidelines and a more tightly defined scope could mostly avoid these problems and would enable its application at this point in time. Extensive testing of the behaviour of a wide range of standard Java libraries in MINHA's simulated executions is needed to provide more confidence that students will not encounter undocumented issues and can be sufficiently informed of potential pitfalls. Alternatively, lower level

protocols of the GCS stack can be replaced with simulated counterparts to allow for a local simulation of distributed executions. Such a modification can be straightforward if all timing references are explicitly handled through the framework rather than using language features[70], or the simulated counterparts are real-time capable.

4.3 Future Work

Many interesting questions have been raised in this thesis that warrant a more in-depth discussion. From a computing education perspective the development of educational tools for distributed computing and other, more advanced computer science topics needs to be addressed, as there is currently little available literature in this area. Once the developed *e*GCS is finalized it will have to be used in a realistic educational setting and be compared to other tools so its educational effectiveness can be evaluated. In (3.3.1) it has also been outlined that currently visualization is not considered and will have to be included at a later stage. The topic of visualization leads to many questions in regards to its provision for an *e*GCS. Besides employing data visualization it is unclear how the complex algorithms of the various protocols found in a (*e*)GCS stack could effectively and possibly automatically be visualized to aid in the learning process.

The bytecode instrumentation approach of MINHA highlights an interesting possibility for augmenting regular student code with educationally beneficial features. It offers the ability to include additional code in a transparent manner, removing from learners the burden of having to either use or include a special API to receive these benefits. There are instances of educational tools using bytecode instrumentation to augment regular program code, especially in regards to visualizing the command flow ([LZ97],[WBE10,[TSH05],[RLBM11],[ESCS12],[SB08]) but to the author's knowledge this mechanism is not further explored. Through its goal of allowing multiple JVMs to be executed locally in a single JVM while simulating key components, MINHA transforms the targeted code so its execution is synchronized with the discrete event simulation. Such a property, in combination with the ability to define which language components are replaced by simulated counterparts, potentially renders MINHA a powerful framework for developing educational software tools. Future work needs to address the problems encountered during the prototype development if MINHA is to be used in such an educational context. Specifically, its correct behaviour needs to be ensured for a wide range of libraries and language features to avoid exposing learners to framework specific, hard to debug issues that may be less problematic for an experienced developer.

In respect to the presented *e*GCS prototype several key aspects need to be addressed in future work. Due to time constraints only an initial prototype could be developed which will need to be finalized. Once the *e*GCS is completed formal specifications and rigorous testing of the defined properties need to be done. Another important issue is the integration of an alternative simulation approach until the described problems in regards to MINHA have been addressed. The design of the original ADS laboratory simulator and also the presented use case implementation (see 3.5.7) require students to implement asynchronous protocols so no additional timing considerations need to be made. This allows for an easy specification of deterministic failure scenarios. For more difficult

[70]Such as using the system clock for timing.

problems such as Consensus and therefore group membership and virtual synchrony a mode of partial synchrony is unavoidable if these problems are to be solvable. In ([CT96]) Chandra and Toueg already point out the advantages of relying on a failure detector abstraction instead of using concrete timings. An interesting approach in this respect would be to have all protocols rely exclusively on such a failure detector abstraction for any timing assumptions while also tying the failure detector implementations into the simulation model.

Another interesting extension to the current design may be to employ the generic GCS API *jGCS* ([CPR06]) as a front-end for the *e*GCS prototype. jGCS offers bindings for several group communication systems such as JGroups, Appia and Spread and could be used in the second part of the laboratory so students can easily switch between their own implementation based on the *e*GCS prototype and other GCSs.

Literature

[ACBMT95] Anceaume, Emmanuelle, et al. *On the formal specification of group membership services*. Cornell University, 1995.

[AH08] Arad, Cosmin, and Seif Haridi. "*Practical protocol composition, encapsulation and sharing in kompics."* Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on. IEEE, 2008.

[Ara13] Arad, Cosmin Ionel. "Programming Model and Protocols for Reconfigurable Distributed Systems." (2013).

[AS98] Amir, Yair, and Jonathan Stanton. *The spread wide area group communication system*. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[Ban98a] Ban, Bela. "Design and implementation of a reliable group communication toolkit for java." *Cornell University* (1998).

[Ban98b] Ban, Bela. "JavaGroups–Group Communication Patterns in Java." *Department of Computer Science Cornell University* (1998): 1-18.

[BCT96] Basu, Anindya, Bernadette Charron-Bost, and Sam Toueg. "Simulating reliable links with unreliable links in the presence of process crashes." *Distributed algorithms*. Springer Berlin Heidelberg, 1996. 105-122.

[BDGB94] Babaoglu, Ozalp, Renzo Davoli Luigi-Alberto Giachini, and Mary Baker. "Relacs: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems." (1994).

[Ben-O83] Ben-Or, Michael. "Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols." *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983.

[BFM+06] Bünzli, Daniel, et al. "Advances in the design and implementation of group communication middleware." *Dependable Systems: Software, Computing, Networks*. Springer Berlin Heidelberg, 2006. 172-194.

[BJ87] Birman, Ken, and Thomas Joseph. *Exploiting virtual synchrony in distributed systems*. Vol. 21. No. 5. ACM, 1987.

[BM03] M. Bauderon and M. Mosbah. 2003. A Unified Framework for Designing, Implementing and Visualizing Distributed Algorithms. *Electronic Notes in Theoretical Computer Science 72 No. 3*

[BMN05] Bünzli, Daniel C., Sergio Mena, and Uwe Nestmann. "Protocol composition frameworks a header-driven model." *Network Computing and Applications, Fourth IEEE International Symposium on*. IEEE, 2005.

[BS99] Mordechai Ben-Ari and Shawn Silverman. 1999. DPLab: an environment for distributed programming. *SIGCSE Bull.* 31, 3 (June 1999), 91-94.

[BK56] Bloom, Benjamin Samuel, and David R. Krathwohl. "Taxonomy of educational objectives: The classification of educational goals. Handbook I: Cognitive domain." (1956).

[Ben-A01] Mordechai Ben-Ari. 2001. Interactive execution of distributed algorithms. *J. Educ. Resour. Comput.* 1, 2es, Article 2 (August 2001). DOI=10.1145/384055.384057 http://doi.acm.org/10.1145/384055.384057

[BGH+01] Brasileiro, Francisco, et al. "Eva: an event-based framework for developing specialised communication protocols." *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*. IEEE, 2001.

[BHvR05] Barr, Rimon, Zygmunt J. Haas, and Robbert van Renesse. "JiST: An efficient approach to simulation using virtual machines." *Software: Practice and Experience* 35.6 (2005): 539-576.

[Bir05] Kenneth P. Birman. 2005. Reliable Distributed Systems.

[Bir10] Birman, Ken. "A history of the virtual synchrony replication model." *Replication*. Springer Berlin Heidelberg, 2010. 91-120.

[BR01] C. Burger and K. Rothermel. 2001. A framework to support teaching in distributed systems. *J. Educ. Resour. Comput.* 1, 1es, Article 3 (March 2001).

[BSS91] Birman, Kenneth, Andre Schiper, and Pat Stephenson. "Lightweight causal and atomic group multicast." *ACM Transactions on Computer Systems (TOCS)* 9.3 (1991): 272-314.

[BWWZ05] Burri, Nicolas, et al. "SANS: A simple ad hoc network simulator." *Proceedings of the Conference on Educational Media, Hypermedia, and Telecommunications (ED-Media), Montreal, Canada*. 2005.

[Bru11] Erik Brunvand. 2011. Games as motivation in computer design courses: I/O is the key. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (SIGCSE '11). ACM, New York, NY, USA, 33-38. DOI=10.1145/1953163.1953178 http://doi.acm.org/10.1145/1953163.1953178

[Cri91] Cristian, Flaviu. "Reaching agreement on processor-group membrship in synchronous distributed systems." *Distributed Computing* 4.4 (1991): 175-187.

[CBCP11] Carvalho, Nuno A., et al. "Experimental evaluation of distributed middleware with a virtualized java environment." *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, 2011.

[CCR+03] Chun, Brent, et al. "Planetlab: an overlay testbed for broad-coverage services." *ACM SIGCOMM Computer Communication Review* 33.3 (2003): 3-12.

[CDK+11] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Distributed Systems Concepts and Design Fifth Edition.

[CGR11] Cachin, Christian, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer, 2011.

[CHS+03] Marco Cicolini, Beatrice Huber, Simon Schlachter and Micha Trautweiler. Entwicklung eines rasanten Peer-to-Peer Spieles. April 2003

[CHTCB95] Chandra, Tushar Deepak, et al. "On the impossibility of group membership." *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996.

[CKP+04] Ioannis Chatzigiannakis, Athanasios Kinalis, Athanasios Poulakidas, Grigorios Prasinos, and Christos Zaroliagis. 2004. DAP: A Generic Platform for the Simulation of Distributed Algorithms. In *Proceedings of the 37th annual symposium on Simulation* (ANSS '04). IEEE Computer Society, Washington, DC, USA, 167-.

[CPR06] Nuno Carvalho, José Pereira and Luís Rodrigues. Towards a Generic Group Communication Service. In Proc. of the International Symposium on Distributed Objects and Applications (DOA). 2006

[CSCS] Edwards, Stephen H., et al. "Running students' software tests against each others' code: new life for an old gimmick." *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012.

[CT96] Chandra, Tushar Deepak, and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems." *Journal of the Association for Computing Machinery* 43.2 (1996).

[CL98] José C. Cunha and João Lourenço. 1998. An integrated course on parallel and distributed processing. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education* (SIGCSE '98), Daniel Joyce and John Impagliazzo (Eds.). ACM, New York, NY, USA, 217-221.

[Dav07] Moti David. 2007. Interactive Execution of Distributed Algorithms.

[Der07] Bilel Derbel. 2007. A Brief Introduction to ViSiDiA

[Déf00] Défago, Xavier. *Agreement-related problems: From semi-passive replication to totally ordered broadcast*. Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2000.

[DSU04] Défago, Xavier, André Schiper, and Péter Urbán. "Total order broadcast and multicast algorithms: Taxonomy and survey." *ACM Computing Surveys (CSUR)* 36.4 (2004): 372-421.

[DH12a] Drejhammar, Frej, and Seif Haridi. "Efficient Simulation of View Synchrony" *SICS Technical Report T2012:07* August 2, 2012 Swedish Institute of Computer Science Box 1263." (2012).

[DH12b] Drejhammar, Frej, and Seif Haridi. "Efficient simulation of view synchrony." *Proceedings of the Winter Simulation Conference*. Winter Simulation Conference, 2012.

[DM96] Dolev, Danny, and Dalia Malki. "The Transis approach to high availability cluster communication." *Communications of the ACM* 39.4 (1996): 64-70.

[DMS94] Dolev, Danny, Dalia Malki, and Ray Strong. *An asynchronous membership protocol that tolerates partitions*. Leibniz Center for Research in Computer Science, Department of Computer Science, Hebrew University of Jerusalem, 1994.

[DLS88] DWORK, C., LYNCH, N. A., AND STOCKMEYER,L. 1988. Consensus in the presence of partial synchrony. J. ***ACM 35, 2*** (Apr.), 288-323.

[DDS87] DOLEV, D., DWORK, C., and STOCKMEYER,L. 1987. On the minimal synchronism needed for distributed Consensus. J. ***ACM*** 34, 1 (Jan.), 77-97.

[EHH+99] Estrin, Deborah, et al. "Network visualization with the VINT network animator nam." *USC Computer Science Department Technical Report* (1999): 99-703.

[Fall99] Fall, Kevin. "Network emulation in the Vint/NS simulator." *Computers and Communications, 1999. Proceedings. IEEE International Symposium on*. IEEE, 1999.

[Fek03] Fekete, Alan. "Using counter-examples in the data structures course." *Proceedings of the fifth Australasian conference on Computing education-Volume 20*. Australian Computer Society, Inc., 2003.

[FKK+05] Farchi, Eitan, et al. "Effective testing and debugging techniques for a group communication system." *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005.

[FR95] Friedman, Roy, and Robbert Van Renesse. "Strong and weak virtual synchrony in Horus." *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on*. IEEE, 1996.

[Fuzz08] Fuzzati, Rachele. *A formal approach to fault tolerant distributed Consensus*. Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2008.

[GG97a] Garbinato, Benoit, and Rachid Guerraoui. *Bast, A Framework for Reliable Distributed Computing*. No. LSR-REPORT-1997-007. 1997.

[GG97b] Garbinato, Benoit, and Rachid Guerraoui. "Using the Strategy Design Pattern to Compose Reliable Distributed Protocols." *COOTS*. 1997.

[GHJV94] Gamma, Erich, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GS01] Guerraoui, Rachid, and André Schiper. "The generic Consensus service." *Software Engineering, IEEE Transactions on* 27.1 (2001): 29-41.

[GHRT01] Greve, Fabiola, et al. "Primary component asynchronous group membership as an instance of a generic agreement framework." *Autonomous Decentralized Systems, 2001. Proceedings. 5th International Symposium on*. IEEE, 2001.

[GlN04] Greve, Fabıola Gonçalves Pereira, and Jean-Pierre Le Narzul. "Designing a configurable group service with agreement components." *Workshop de Testes e Tolerancia a Falhas (WTF 2004), SBRC, Gramado, Brasil*. 2004.

[Gho07] Sukumar Ghosh. 200. Distributed Systems An Algorithmic Approach.

[GMB00] Gruner, Stefan, Mohamed Mosbah, and Michel Bauderon. "A New Tool for the Simulation And Visualization of Distributed Algorithms." (2000).

[Got03] Tim Gottwald. 2003. Entwurf und Implementierung eines Baukastens für Verteilte Algorithmen. *Diplomarbeit, Fachbereich Design und Informatik, Fachhochschule Trier, University of Applied Sciences. (November 2003).*

[GS96] Guerraoui, Rachid, and André Schiper. *Atomic Multicast harder than Atomic Broadcast*. No. LSR-REPORT-1996-008. Technical Report, Ecole Polytechnique Fédérale de Lausanne, Computer Science Department, 1996.

[Hay98] Hayden, Mark Garland. *The ensemble system*. Diss. Cornell University, 1998.

[HB14] Hamzah, Upu, and Bustang Bustang. "CONSTRUCTIVISM VERSUS COGNITIVE LOAD THEORY: IN SEARCH FOR AN EFFECTIVE MATHEMATICS TEACHING." *Proceeding of International Conference On Research, Implementation And Education Of Mathematics And Sciences 2014*. Yogyakarta State University, 2014.

[HD00] Christopher Hundhausen and Sarah Douglas. 2000. Using Visualizations to Learn Algorithms: Should Students Construct Their Own, or View an Expert's?. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)* (VL '00). IEEE Computer Society, Washington, DC, USA, 21-.

[HDS01] Christopher D. Hundhausen, Sarah A. Douglas and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. 2001.

[Her91] Herlihy, Maurice. "Wait-free synchronization." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991): 124-149.

[Hol91] Holzmann, Gerard J. "DESIGN AND VALIDATION OF COMPUTER PROTOCOLS." (1991).

[HP91] Hutchinson, Norman C., and Larry L. Peterson. "The x-kernel: An architecture for implementing network protocols." *Software Engineering, IEEE Transactions on* 17.1 (1991): 64-76.

[HRSD+] Hibler, Mike, et al. "Large-scale Virtualization in the Emulab Network Testbed." *USENIX Annual Technical Conference*. 2008.

[HT94] Hadzilacos, Vassos, and Sam Toueg. "A modular approach to fault-tolerant broadcasts and related problems." (1994).

[HMRT99] Hurfin, Michel, et al. "A general framework to solve agreement problems." *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*. IEEE, 1999.

[Hunt10] John M. Hunt. 2010. Nifty assignment: concurrent multi-user battleship. *J. Comput. Small Coll.* 26, 2 (December 2010), 215-219.

[JFR93] Jahanian, Farnam, Sameh Fakhouri, and Ragunathan Rajkumar. "Processor group membership protocols: Specification, design and implementation." *Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on*. IEEE, 1993.

[Kol99] Boris Koldehofe. 1999. Animation and Analysis of Distributed Algorithms. May 1999.

[Kol05] Boris Koldehofe. 2005. Distributed Algorithms and Educational Simulation/Visualisation in Collaborative Environments. *Dissertation, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University.* ISBN 91-7291-572-2, Göteborg, Sweden.

[KP94] Yvon Kermarrec and Laurent Pautet. 1994. Ada Reusable Software Components for Teaching Distributed Systems. In *Proceedings of the 7th SEI CSEE Conference on Software Engineering Education*, Jorge L. Díaz-Herrera (Ed.). Springer-Verlag, London, UK, 77-96.

[KPT03] Koldehofe, Boris, Marina Papatriantafilou, and Philippas Tsigas. "Integrating a simulation-visualisation environment in a basic distributed systems course: a case study using LYDIAN." *ACM SIGCSE Bulletin*. Vol. 35. No. 3. ACM, 2003.

[KPT06] Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. 2006. LYDIAN: An extensible educational animation environment for distributed algorithms. *J. Educ. Resour. Comput. 6, 2, Article 1* (June 2006).

[KSC06] Kirschner, Paul A., John Sweller, and Richard E. Clark. "Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching." *Educational psychologist* 41.2 (2006): 75-86.

[KT91] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222-230, Arlington, TX, May 1991.

[Kur09] Stan Kurkovsky. 2009. Can mobile game development foster student interest in computer Science?. In *Games Innovations Conference (ICE-GIC), 2009*. London

[Lam84] Lamport, Leslie. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6.2 (1984): 254-280.

[Lam98] Lamport, Leslie. "The part-time parliament." *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998): 133-169.

[LBM11] Lönnberg, Jan, Mordechai Ben-Ari, and Lauri Malmi. "Java replay for dependence-based debugging." *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM, 2011.

[LSP82] Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982): 382-401.

[LZ97] Lee, Han B., and Benjamin G. Zorn. "Bytecode Instrumentation as an Aid in Understanding the Behavior of Java Persistent Stores." *OOPSLA 1997 Workshop on Garbage Collection and Memory Management*. 1997.

[Mal96] Malloth, Christoph Peter. *Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks*. Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 1996.

[MPS91] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Proceedings of the IEEE International Working Conference on Dependable Computing For Critical Applications*, pages 137-145, Tucson, AZ, February 1991.

[MB02] Hisham H. Muhammad and Marinho P. Barcellos. 2002. Simulating Group Communication Protocols Through an Object-Oriented Framework. In *Proceedings of 35th SCS Annual Simulation Simposium.* April 2002

[MB03] Muhammad, Hisham H., and Marinho P. Barcellos. "Protocol Simulation with the Simmcast Framework." *XXI Simpósio Brasileiro de Redes de Computadores (SBRC2003), Anais, Salão de Ferramentas, SBC, Natal* (2003): 889-896.

[MFF+97] McCanne, Steven, et al. "Network simulator ns-2." (1997): 1059-1068.

[MAMA94] Moser, Louise E., et al. "Extended virtual synchrony." *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE, 1994.

[MSMA94] Moser, Louise E., P. M. Melliar-Smith, and Vivek Agrawala. "Processor membership in asynchronous distributed systems." *Parallel and Distributed Systems, IEEE Transactions on* 5.5 (1994): 459-473.

[Men06] Mena de la Cruz, Sergio. "Protocol composition frameworks and modular group communication." (2006).

[MN99] Mehlhorn, Kurt and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[Meh01] Mehringer, John. "The NAM editor." *A presentation for the CONSER retreat, slide* 2 (2001).

[Mish02] Shivakant Mishra. 2002. A Peer-to-Peer System Assignment for Teaching Distributed Systems Concepts. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 4* (PDPTA '02), Hamid R. Arabnia (Ed.), Vol. 4. CSREA Press 2002-2005.

[Mor05] Thiemo Morth. 2005. Weiterentwicklung eines nachrichtengesteuerten Programms zur Simulation verteilter Algorithmen. *Projektarbeit, Fachbereich Design und Informatik, Fachhochschule Trier, University of Applied Sciences.* (February 2005).

[NFmN+02] Naps, Thomas L., et al. "Exploring the role of visualization and engagement in computer science education." *ACM SIGCSE Bulletin*. Vol. 35. No. 2. ACM, 2002.

[O'Don06] Fionnuala O'Donnell. 2006. Simulation Frameworks for the Teaching and Learning of Distributed Algorithms. *Dissertation, University of Dublin, Trinity College.* (February 2006).

[OPSS94] Oki, Brian, et al. "The Information Bus: an architecture for extensible distributed systems." *ACM SIGOPS Operating Systems Review*. Vol. 27. No. 5. ACM, 1994.

[OG05] Rainer Oechsle and Tim Gottwald. 2005. DisASTer (distributed algorithms simulation terrain): a platform for the implementation of distributed algorithms. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '05). ACM, New York, NY, USA, 44-48. DOI=10.1145/1067445.1067461 http://doi.acm.org/10.1145/1067445.1067461

[PB01] Stella Papakosta and Cora Burger. 2001. Generating Interactive Protocol Simulations and Visualizations for Learning Environments. In Proceedings of the CaberNet: 4th Plenary Workshop, June 2001.

[PSE+05] Pears, Arnold, et al. "Constructing a core literature for computing education research." *ACM SIGCSE Bulletin* 37.4 (2005): 152-161.

[vRBC+93] Van Renesse, Robbert, et al. "The horus system." *Reliable Distributed Computing with the Isis Toolkit* (1993): 133-147.

[Var01] Varga, András. "The OMNeT++ discrete event simulation system." *Proceedings of the European Simulation Multiconference (ESM'2001)*. Vol. 9. sn, 2001.

[PT98] Marina Papatriantafilou and Philippas Tsigas. 1998. Towards a Library of Distributed Algorithms and Animations (Lydian). 1998

[Piy11] Rody Piyasin Unpublished thesis! (permission of author to reference)

[RB91] Ricciardi, Aleta M., and Kenneth P. Birman. "Using process groups to implement failure detection in asynchronous environments." *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*. ACM, 1991.

[RB93] Ricciardi, Aleta M., and Kenneth P. Birman. *Process membership in asynchronous environments*. Cornell University, 1993.

[RR00] Gilad Ravid and Sheizaf Rafaeli. 2000. Multi Player, Internet and Java-based Simulation Games: Learning and Research in implementing a Computerized Version of the „Beer-Distribution Supply Chain Game“. In Proceedings of the International Conference on Web-Based Modeling & Simulation (WEBSIM 2000), San Diego, CL.

[RWS06] Rütti, Olivier, Paweł T. Wojciechowski, and André Schiper. "Service interface: a new abstraction for implementing and composing protocols." *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006.

[Rüt09] Rütti, Olivier. *Concurrency and dynamic protocol update for group communication middleware*. Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2009.

[SB08] Sundararaman, Jaishankar, and Godmar Back. "HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java." *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 2008.

[Schr02] Wolfgang Schreiner. 2002. A java toolkit for teaching distributed algorithms. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education* (ITiCSE '02). ACM, New York, NY, USA, 111-115.

[SCW+10] Sahar S. Shabanah, Jim X. Chen, Harry Wechsler, Daniel Carr, and Edward Wegman. 2010. Designing Computer Games to Teach Algorithms. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations* (ITNG '10). IEEE Computer Society, Washington, DC, USA, 1119-1126.

[Stä04] Bernhard Stähli. 2004. MultiSweeper Ein fesselndes Multiplayer-Spiel auf P2P-Basis. Sept 2004.

[ST06] Schiper, Andre, and Sam Toueg. "From set membership to group membership: A separation of concerns." *Dependable and Secure Computing, IEEE Transactions on* 3.1 (2006): 2-12.

[STP+92] Spirakis, P., et al. "Distributed system simulator (DSS)." *STACS 92*. Springer Berlin Heidelberg, 1992. 615-616.

[SRSD08] Song, Yee Jiun, et al. "The building blocks of Consensus." *Distributed Computing and Networking*. Springer Berlin Heidelberg, 2008. 54-72.

[Sung09] Kelvin Sung. 2009. Computer games and traditional CS courses. *Commun. ACM* 52, 12 (December 2009), 74-78. DOI=10.1145/1610252.1610273 http://doi.acm.org/10.1145/1610252.1610273

[Swe88] Sweller, John. "Cognitive load during problem solving: Effects on learning." *Cognitive science* 12.2 (1988): 257-285.

[TADM06] Taylor, Ian, et al. "Agentj: Enabling Java NS-2 simulations for large scale distributed multimedia applications." *Distributed Frameworks for Multimedia Applications, 2006. The 2nd International Conference on*. IEEE, 2006.

[TS07] Andrew S. Tanenbaum and Maarten Van Steen. 2007. Distributed Systems: Principles and Paradigms 2nd Edition.

[TSH05] Tilevich, Eli, Yannis Smaragdakis, and M. Handle. "Appletizing: Running legacy Java code remotely from a Web browser." *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005.

[USD01] Peter Urban, Andre Schiper, and Xavier Defago. 2001. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In *Proceedings of the The 15th International Conference on Information Networking* (ICOIN '01). IEEE Computer Society, Washington, DC, USA, 503-.

[Val02] Valcke, Martin. "Cognitive load: updating the theory?." *Learning and Instruction* 12.1 (2002): 147-154.

[WBE10] Woods, Michael, Godmar Back, and Stephen Edwards. "An infrastructure for teaching CS1 in the cloud." *ASEE Southeast Section Annual Conference*. 2010.

[WHS01] Wong, Gary T., Matti A. Hiltunen, and Richard D. Schlichting. "A configurable and extensible transport protocol." *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 1. IEEE, 2001.

[WSHW08] Weingärtner, Elias, et al. "Synchronized network emulation: matching prototypes with complex simulations." *ACM SIGMETRICS Performance Evaluation Review* 36.2 (2008): 58-63.

[WSvL+11] Weingärtner, Elias, et al. "SliceTime: A Platform for Scalable and Accurate Network Emulation." *NSDI*. 2011.

[WRSW10] Weingartner, E., et al. "Flexible analysis of distributed protocol implementations using virtual time." *Software, Telecommunications and Computer Networks (SoftCOM), 2010 International Conference on*. IEEE, 2010.

[WKC+09] Joel Wein, Kirill Kourtchikov, Yan Cheng, Ron Gutierez, Roman Khmelichek, Matthew Topol, and Chris Sherman. 2009. Virtualized games for teaching about distributed systems. In *Proceedings of the 40th ACM technical symposium on Computer science education* (SIGCSE '09). ACM, New York, NY, USA, 246-250.

[WMB+06] Robert J. Walters, David E. Millard, Philip Bennet, David Argles, Stephen Courch, Lester Gilbert and Gary Wills. 2006. Teaching the Grid: Learning Distributed Computing with the M-grid Framework. *In Proceedings of the 2006 World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA 2006), pp.* 2234-2241. Southampton, UK.

[WMK95] Whetten, Brian, Todd Montgomery, and Simon Kaplan. *A high performance totally ordered multicast protocol*. Springer Berlin Heidelberg, 1995.

[WRS04] Wojciechowski, Paweł, O. Riitti, and André Schiper. "SAMOA: framework for synchronisation augmented microprotocol approach." *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004.

[You07] G. Michael Youngblood. 2007. Using XNA-GSE game segments to Engage Students in Advanced Computer Science Education. In *Proceedings of the Microsoft Academic Days on Game Development in Computer Science Education.* pp. 20-26

Internet references

[FG11] Froihofer, Lorenz and Karl Göschka. “ADS – Advanced Distributed Systems – Laborübung”. 05 May 2011. <http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/uebung/>.

[GF11] Göschka, Karl M. and Lorenz Froihofer. “ADS – Advanced Distributed Systems – Vienna University of Technology”. 17 April 2011. <http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/>.

[Cach08] Cachin, Christian. “View-synchronous Group Communication“. Security and Fault-tolerance in Distributed Systems. IBM Zurich Research Laboratory. Spring 2008. Web. 20 Nov. 2012 <http://resist.isti.cnr.it/free_slides/security/cachin/viewsync.pdf/>

[Fow04] Martin Fowler. “Inversion of Control Containers and the Dependency Injection pattern”. 23 January 2004. Web. 25 April 2013 <http://martinfowler.com/articles/injection.html/>

[Web13] Bernhard Weber. “Alcatraz – Auf zum rettenden Fluchtboot!”. Web. 3 April 2013. <http://www.bernhardweber.de/alcatraz.html/>

[Pic11] PicoContainer. 13 July 2011. Web. 6 December 2012. <http://picocontainer.com/>