A Modular Framework to Implement Fault Tolerant Distributed Services

by

P. Nicolas Kokkalis

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

A modular framework to implement fault tolerant distributed services

P. Nicolas Kokkalis

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

In this thesis we present a modular architecture and an implementation for a generic fault-tolerant distributed service which, broadly speaking, is based on Lamport's state machine approach.

An application programmer can develop client applications under the simplifying assumption that the service is provided by a single, reliable server. In reality, the service is provided by replicated, failure-prone servers. Our architecture presents to the application the same interface as the ideal single and reliable server, and so the application can be directly plugged into it. A salient feature of our architecture is that faulty replicated servers are dynamically replaced by correct ones, and these changes are transparent to the clients. To achieve this, we use an idea proposed in [13]: the same atomic broadcast algorithm is used to totally order both the client's requests and the requests to change a faulty server, into a single commonly-agreed sequence of requests.

To my first academic advisor, Manolis G.H. Katevenis, who introduced me to research.

Acknowledgements

I'm profoundly indebted to Vassos Hadzilacos for his thoughtful supervision, and to Sam Toueg for helpful suggestions, and many challenging discussions. Thanks as well to George Giakkoupis for his valuable comments and assistance editing a late draft of this thesis. I would also like to thank my family, my friends and the grspam for their continuous support and for keeping my spirit up.

Contents

Chapter 1

Introduction

1.1 Goals of the Thesis

Consider the following distributed client-server application. A server maintains an integer register R, and a set of clients perform asynchronously read and update operations on R. The server processes the requests sequentially in order of arrival. In this application, the programmer basically has to design two *modules*, i.e., pieces of software: the *client application module* and the *server application module*. The client application module should interact with the end users and send their requests to the server application module. It should also receive the replies generated by the server application module and deliver them to the appropriate end users. The server application module is responsible for processing the requests sent by the client application modules, and for replying to each client request as required.

Clearly, the single server approach is susceptible to server failures. A standard way to improve robustness to server failures is by the use of replication as follows. The single server is replaced by a set of servers that collectively emulate a single server, in the sense that each server maintains a copy of R and performs on R the same sequence of operations that the single server would perform. Moreover different servers must perform the same sequence of operations. Such an approach would allow the system to tolerate the failure of a portion of the servers. However, it introduces additional complexity in the design of the application. In particular, maintaining consistency and managing coordination among servers is a non-trivial task. It would be nice if we could separate the issues of reliability from the actual application.

The goal of this thesis is the design and implementation of an application-independent middleware/framework that is responsible for seamless server replication transparently to the application. In other words, the application programmer should design a client-server application assuming a single server model where the server is not allowed to crash. He should also implement and test the application in a single server environment. When these steps are completed, the resulting client and server application modules can be readily plugged into the framework, which takes care of fault tolerance. We name our architecture *nofaults.org* or simply *nofaults*.

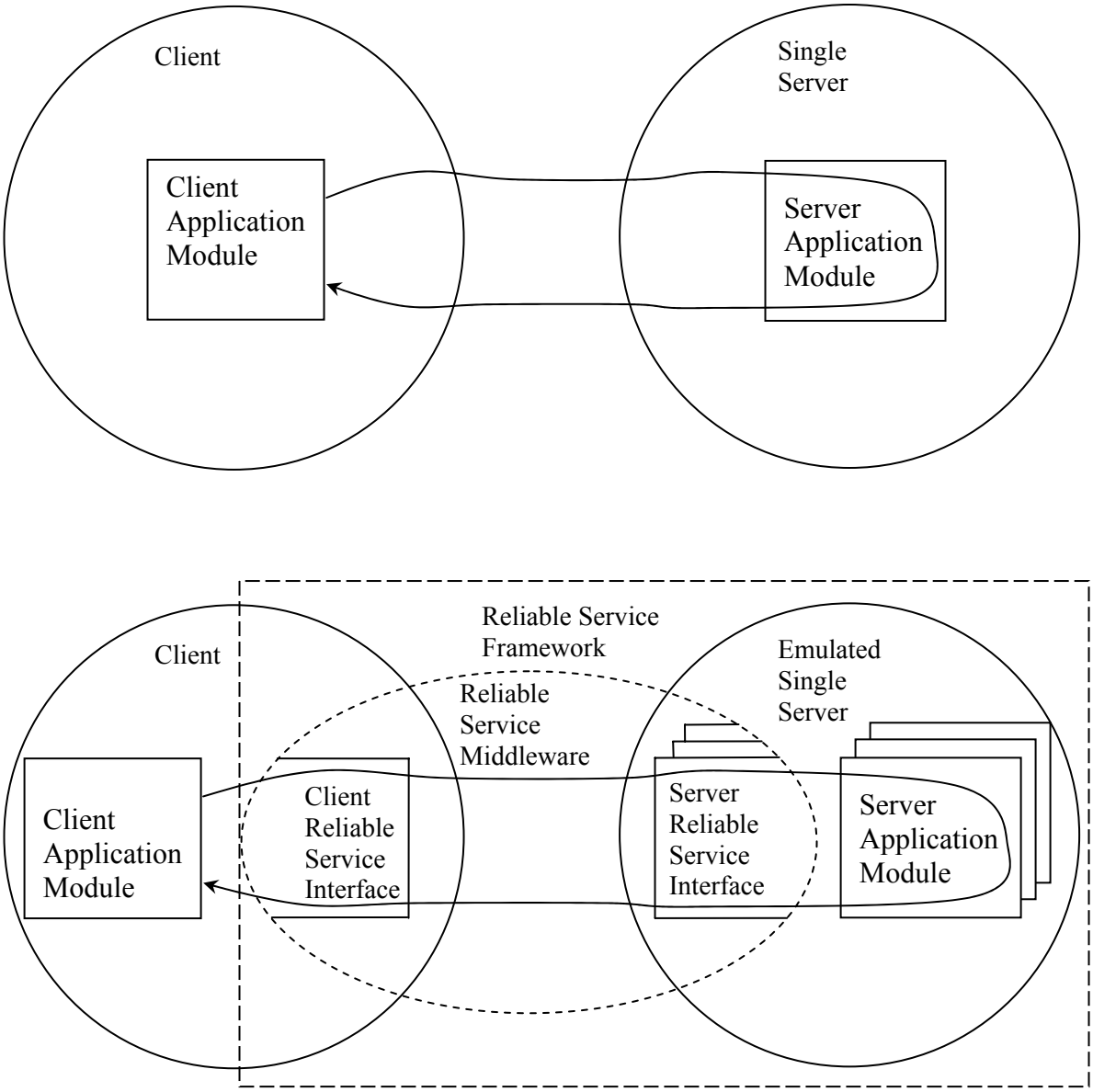

Figure 1 (a) The application is designed, implemented, and tested in a simple single server model (b) Once the application is ready, it is plugged into the reliable service framework and automatically becomes fault tolerant.

1.2 Organization of the Thesis

In Section 2, we provide some background information on the concepts of distributed computing that we use in our system. In Section 3, we describe the overall structure of our system, briefly present the internal organization of each process of the system, and describe how information flows in the system. For the shake of clarity in Section 4 we present a simple version of the system that uses a static set of server processes. In Section 5 we show how this simple version presented can be extended to become a more sophisticated version which allows faulty servers to be dynamically replaced by backup standby processes. Finally, in Section 6 we present conclusions and discuss ideas for future work.

Chapter 2

Related Work – Background

The *state machine approach* is a general method for achieving fault tolerant services and implementing decentralized control in distributed systems [9]. It works by replicating servers and coordinating client interactions with server replicas. A *state machine* consists of *state variables*, which encode its state, and *requests* which transform its state. Each request is implemented by a deterministic program; and the execution of the request is atomic with respect to other requests and modifies the state variables and/or produces some output. A *client* of the state machine forms and submits requests. Lamport presented an algorithm which ensures that all requests are executed in the same order by all the server replicas. This present to the clients the illusion of a single server. The approach we take in this thesis follows this simple but powerful idea [9].

Consensus is the archetypical agreement problem, and agreement lies at the heart of many tasks that arise in fault tolerant distributed computing, such as Atomic Broadcast [11] and Atomic Commit [15, 14, 16, 12]. The consensus problem can be informally described as follows: Each process proposes a value and all the non-crashed processes have to agree on a common value, which should be one of the proposed values. This description of consensus allows *faulty* processes (i.e., processes that crash) to adopt different value than the value *correct*, processes (i.e., processes that do not crash) adopt. In this thesis, we are interested in a more advanced version of Consensus called *Uniform Consensus* that does not allow this to happen, i.e., if a faulty process adopts a value then this value is the same as the value that correct processes adopt.

Solving the consensus problem in asynchronous distributed systems, even when only one process can crash, was proven to be impossible by Fischer, Lynch, and Paterson [4]. To overcome this result many researchers have worked on defining a set a minimum properties that, when satisfied by the runs of a distributed system, makes the problem of consensus solvable [1,5].

A major technique to overcome the impossibility of consensus is by using *Unreliable Failure Detectors*, introduced by Chandra and Toueg [1]. The failure detector service consists of a collection of modules, each of which is associated with one process of the system. Each local failure detector module provides the associated process with some information about failures, often in the form of a list of processes it suspects to have crashed. This information can be incorrect, for example, by erroneously suspecting a

correct process, or by not suspecting a process that has actually crashed. Chandra and Toueg [1] define several classes of failure detectors according to the guarantees they provide. For the measurement of these guarantees *Completeness* and *Accuracy* properties are defined. A Completeness property describes the degree to which the failure detector fails to identify processes that have crashed; whereas, an Accuracy property describes the degree to which the failure detector can erroneously suspect as crashed processes that are actually alive. In addition, in terms of when it is satisfied an Accuracy property is classified as *Perpetual*, if it has to be permanently satisfied, or *Eventual*, if it suffices to be satisfied after some time.

Two of the most important classes of failure detectors are denoted *S* and ◊*S*. Both satisfy the same completeness property (Strong Completeness): Eventually, every process that crashes is permanently suspected by every correct process. S satisfies the following accuracy property (Perpetual Weak Accuracy): Some correct process is never suspected. ◊*S* satisfies the same accuracy property but only eventually (Eventual Weak Accuracy): There is a time after which some correct process is never suspected.

Chandra and Toueg [1] proposed a Consensus protocol that works with any failure detector of class S, and can tolerate up to n-1 failures, where n the number of participating processes. In practice, however, the construction of failure detectors in this class requires strong assumptions about synchrony of the underlying system. On the contrary, failure detectors in class ◊S seem to be easier to implement. Also, there are other compelling reasons to use ◊*S* instead of *S*: If the synchrony assumptions on the

underlying system are violated, an *S*-based algorithm can violate safety (agreement) whereas a ◊*S*-based algorithm will never violate safety: it may only violate liveness. So, if the synchrony assumptions are later restored, the ◊*S*-based algorithm will resume correct operation, while the S-based algorithm is already doomed.

Much work has been done on consensus protocols that rely on ◊S failure detectors: Chandra and Toueg [1], Schiper [6], Hurfin and Raynal [7], Mostefaoui and Raynal [8] proposed some algorithms. All of the above protocols require n>2t, i.e. that the majority of processes are correct. Chandra and Toueg proved that this condition is necessary for any consensus algorithm that uses ◊*S*. Therefore the above protocols are optimal in terms of resiliency. Chandra, Hadzilacos and Toueg proved that ◊*S* is the weakest class of failure detectors that can solve consensus with a majority of correct processes [5].

Chapter 3

System High Level Description

3.1 System Overview

As we mentioned in the introduction, to tolerate server failures the server application module is replicated across several physical servers. Our goal is to hide the details of the replication from both the client and server application modules. This is achieved by inserting an intermediate software module, between the application module and the network in each client and server. On the server side, this intermediate module communicates with the other intermediate modules and provides the illusion of a single server to its local server application module. Similarly, on the client side, the intermediate module communicates with other server intermediate modules and acts as a single server to its client application. In this work, our main interest is the design of generic intermediate modules that work regardless of the actual application modules. From this point on, we will refer to a combination of such intermediate modules with an arbitrary

pair of client/server application modules as a *reliable multi-server system*, or simply *system*.

To emulate a single server, the servers run a consensus protocol. Such protocols, however, are not trivial and their complexity depends (linearly, and for some algorithms quadratically) on the number of participating processes (servers). Therefore, we would prefer to run the protocol in relatively few servers without, as a result, sacrificing the level of reliability. In fact, as mentioned above, to tolerate t failures we need 2t+1 servers in a system equipped with a ◊S failure detector. In practice, however, the assumption that at most t processes can crash may be problematic: As soon as a process crashes the system is only t-1 resilient; sooner or later it may reach a 0-resilient state endangering the whole system in the event of an extra process crash.

In this thesis, we use ideas from [13] to alleviate the situation just described, without increasing t, while keeping the number of servers that run the consensus protocol strictly equal to 2t+1. We introduce back-up servers, which we call *Agents*, that are usually not running the consensus protocol, but are equipped accordingly to replace a crashed server. That way the t-resiliency of the system can be preserved over time, by dynamically updating the group of servers with processes from the group of agents. Additionally, if the agents are designed to batch and forward client requests to the actual servers, they can even help reduce the communication directed to the servers and therefore improve the scalability of the system.

In more detail, our approach considers three groups of processes: A set of Clients (**C**), a set of Servers (**S**), and a set of Agents (**A**). The union of **A** and **S** forms the set of all n potential servers and is static. Each of **A** and **S** may change dynamically during the lifetime of the system, in such a way that they remain distinct and the size **S** is always 2t+1. Obviously n should be greater or equal to 2t+1. We say that the set of servers **S** changes the moment t+1 processes have decided that the value of S should change. In addition, we require that at any time at least a majority of processes currently in **S** are correct, i.e., they do not crash while they are in **S**, and there are at least t+1 processes common to any two consecutive instances of **S**.

Each client is aware of the network addresses of all the processes in **A**∪**S**. To submit a request, a client needs to send the request to just one of these processes. In case the client does not receive a response within a predetermined timeout, the client resubmits the same request to a different process in **A**∪**S**. When a server receives a request it communicates it to the other servers. All requests are executed once by each server in the same order. The servers decide on the ordering of the requests by running a consensus protocol. Only the servers contacted by the client are responsible for responding to each request. Note that each client request is executed only once by each server, regardless of how many times it was send by the client, or how many and which servers were contacted by the client. In case the process that receives the client request initially is an agent, it forwards the request to some server. The big picture of our reliable multi-server system is depicted in Figure 2.

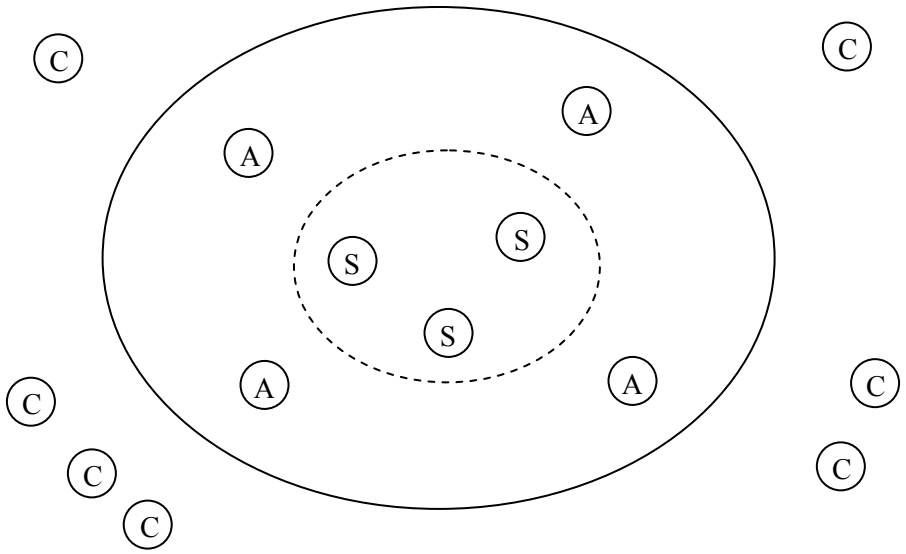

Figure 2 Overview of our reliable multi-server system. Circles labeled by "C" are clients, circles labeled by "A" are agents, and circles labeled by "S" represent servers.

3.2 The model

The system consists of: a dynamic set **S** of 2t+1 ($t \geq 1$) server processes $s_1, s_2, \ldots, s_{2t+1}$; a dynamic set **A** of $m \geq 0$ agent processes $a_1, a_2, \ldots, a_m$; and a dynamic set of k client processes $c_1, c_2, \ldots, c_k$. A process can crash, which means that at any time it may stop taking any further steps and sending any messages. A correct process is one that does not crash. Processes communicate asynchronously through bidirectional channels. There exists one channel for every pair of processes. The channels do not create or alter messages, but are allowed to arbitrarily delay messages. They are not assumed to be FIFO. Moreover the channels can intermittently duplicate or drop messages, but they are

assumed to be fair in the following sense: If a process p sends a message m to a process q infinitely often then q will eventually receive m.

3.3 Layering

In Figures 3-5 we show the architecture of each of the client, server, and agent processes, respectively.

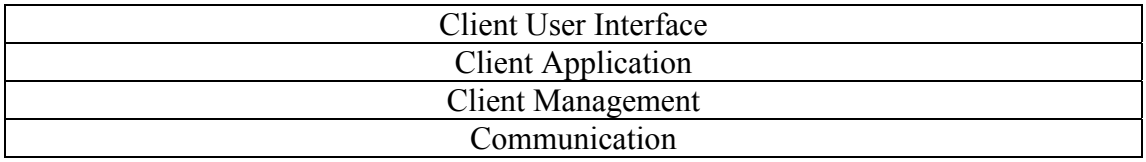

Figure 3 Layers in a Client

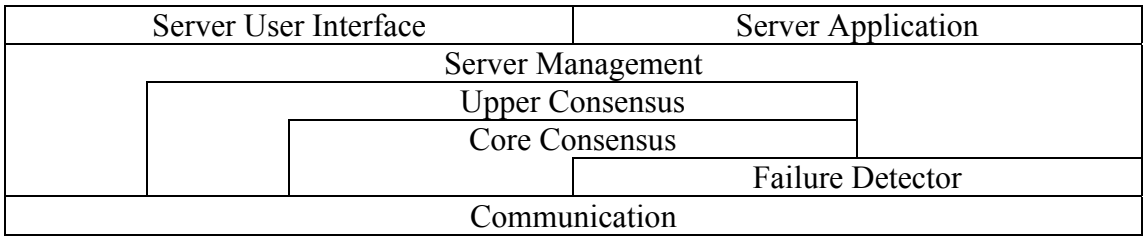

Figure 4 Layers in a Server.

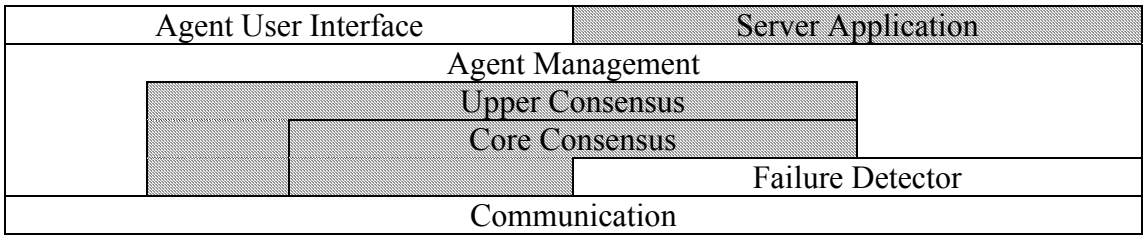

Figure 5 Layers in a Agent. Shaded layer are inactive.

A brief explanation of the layers in the figures above is as follows:

Communication: a custom, protocol agnostic, simple message passing interface that supports reliable non FIFO message transmission.

Failure Detector: provides educated opinion about the correctness of other server/agent processes.

Upper Consensus[1]: a layer responsible for atomically broadcasting messages.

Core Consensus[2]: decides on a total order of the messages of Upper Consensus.

Client Management: hides replication details from the Client Application layer

Server and Agent Management: implements the replication utilizing the layers attached to it, while hiding the replication details from the Server Application layer

Server and Client Application: consists of the server and client application modules

Server, Agent and Client User Interface: interface between the user and the system

A more detailed explanation will be given in following chapters.

3.4 Information flow

The flow of information in our reliable multi-server system is as follows. A user u interacts with the client user interface of a client process c, which informs the Client Application module of c about u's command. The Client Application module examines whether the execution of the command requires the assistance of the Server Application module. If this is the case, the Client Application module generates a request r for the command, and passes r to the Client Management layer of c (by invoking the appropriate interface). Each request has a unique id. Then the Client Management layer of c sends r

[1] A more appropriate name for this layer is "Atomic Broadcast layer"
[2] A more appropriate name for this layer is just "Consensus layer"

to the Server Management layer of some server s (via the Communication layer). The Server Management layer of s passes r to the Upper Consensus layer of s. The Upper Consensus layer of s broadcasts r (through the Communication layer) to the Upper Consensus layers of all the servers. The Upper Consensus layer of each server that receives r passes the unique id of r, id_r, to its local Core Consensus layer. Periodically, as instructed by the Management layer, the Core Consensus layers of the servers run a consensus protocol in order to agree upon a total order of the request ids they have received so far. When the order of id_r is decided, it is passed to the Upper Consensus layer. The order of r is then passed up to the Server Management layer, and eventually to the Server Application layer. In each Server, the Application layer processes r and passes a reply q_r down to the Server Management layer. The Server Management of s sends q_r to the Client Management layer of c, which passes q_r to the Client Application layer. Finally, the Client Application layer of c calls the appropriate function of the Client User Interface of c, informing the user about the result of his action. Figures 6 and 7 illustrate this description of the flow of information is a system with three servers.

In the above description, whenever we say that some layer λ of a process p sends a message m to some layer λ' of a process p′ we mean that the following procedure takes place. Layer λ passes m to the Communication layer of p. Then the Communication layer of p transmits m over the real network to the Communication layer of the destination layer of p′. The Communication layer of p′ passes m to the local layer λ'.

So far we have assumed a failure free execution. We will thoroughly discuss how the system handles failures in subsequent sections. A key point is that the Management layer of every client keeps retransmitting a request to different server/agent processes until it receives a reply. Moreover, the servers and the agents use a mixture of protocols to overcome potential crashes.

Source layer		Destination Layer	Type of information exchanged
Client Application $_c$	→	Client Management $_c$	(Request)
Client Management $_c$	→	Server Management $_s$	(Request)
Server Management $_s$	→	Upper Consensus $_s$	(Request)
Upper Consensus $_s$	→	Upper Consensus $_{all}$	(Request)
Upper Consensus $_{all}$	→	Consensus Core $_{all}$	(RequestID)
Consensus Core $_{all}$	→	Upper Consensus $_{all}$	(Totally-Ordered-RequestID)
Upper Consensus $_{all}$	→	Server Management $_{all}$	(Totally-Ordered-Request)
Server Management $_{all}$	→	Server Application $_{all}$	(Totally-Ordered-Request)
Server Application $_{all}$	→	Server Management $_{all}$	(Reply)
Server Management $_s$	→	Client Management $_c$	(Reply)
Client Management $_c$	→	Client Application $_c$	(Reply)

Figure 6 Information flow in a failure free run of the system.

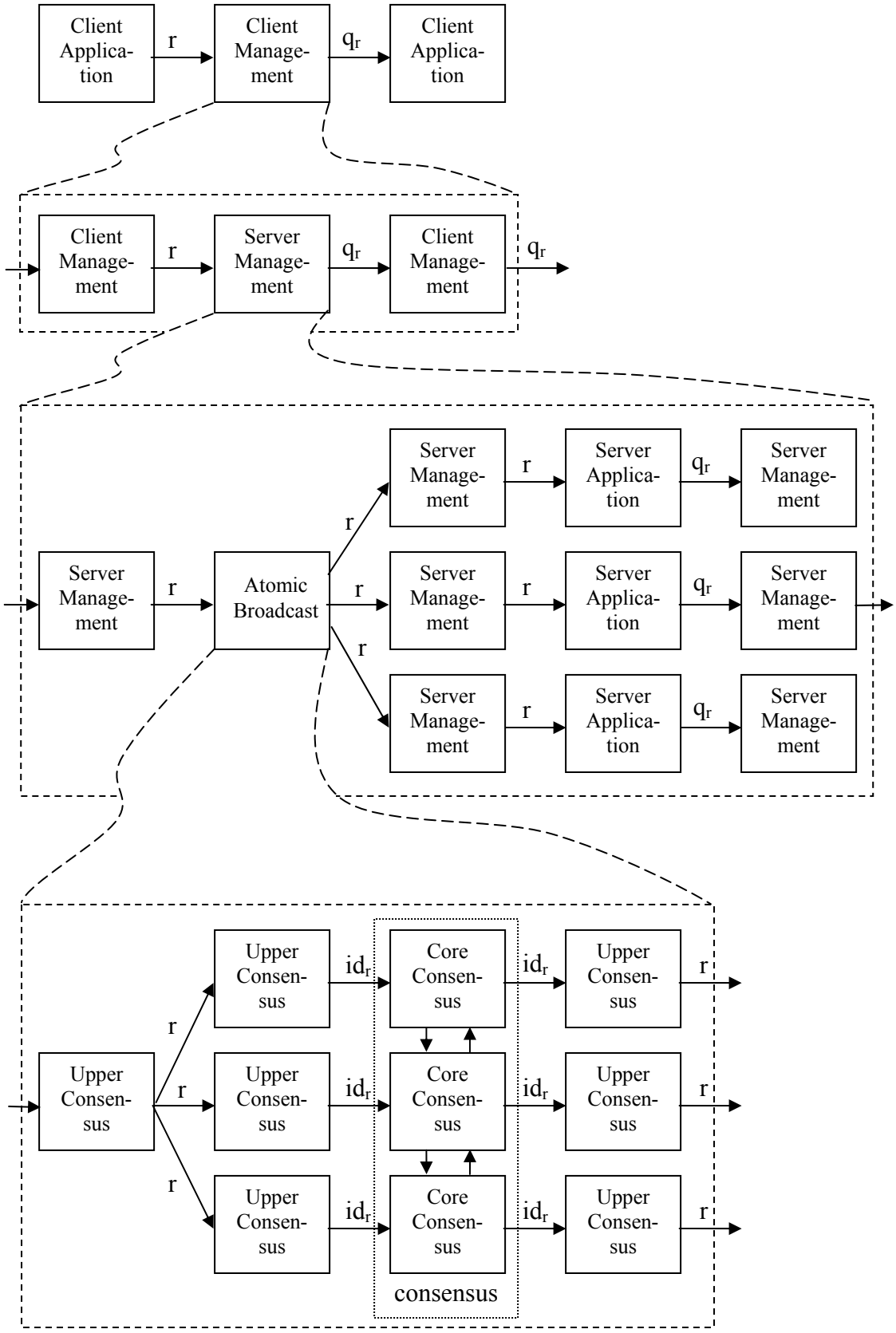

Figure 7 Information flow in a 3-server system.

Chapter 4

The Static Server-Set Version

For the shake of clarity, we will describe two versions of the system. The first one assumes a static set of servers with no agents, while the second one assumes dynamic sets of servers and agents. In section 4 we thoroughly describe the simpler version of static servers, which we cal *the static case*. In section 5 we extend the simplified version to handle changes to the set of servers automatically; we call this version *the dynamic case*.

As mentioned above, we follow a layered system design. Figure 8 and Figure 9 extend Figure 3 and Figure 4, respectively, by displaying the inter-layer communication interfaces. In sections 4.1 – 4.7, we specify the functionality of each layer in bottom-up order.

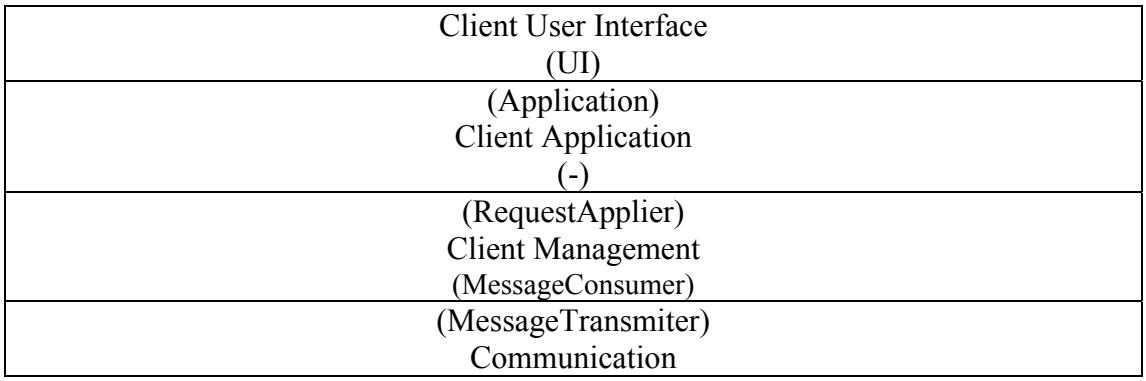

Figure 8 Client Layers. The terms enclosed within brackets denote the interfaces used for communication between layers. The interfaces at the top (bottom) of a box that corresponds to a layer L are used by the layers directly above (below) L to communicate with L. A dash is used when no interface exists for that particular direction. For example, the Client Application layer uses the RequestApplier interface to pass Requests to the Client Management layer.

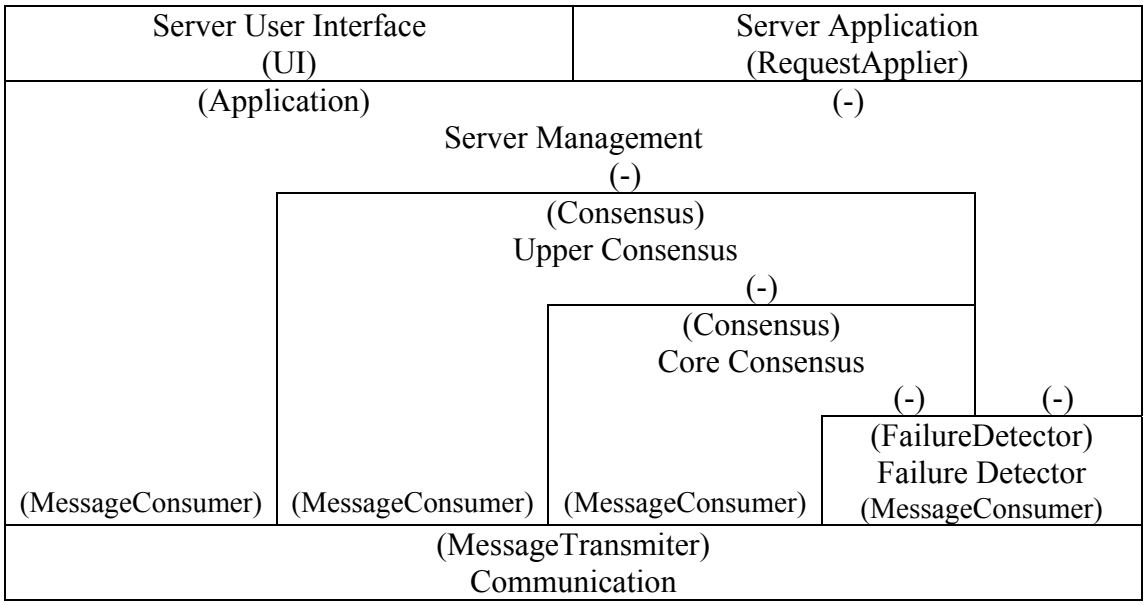

Figure 9 Layers of Server. The terms enclosed within brackets denote the interfaces used for communication between layers. The interfaces at the top (bottom) of a box that corresponds to a layer L are used by the layers directly above (below) L to communicate with L. A dash is used when no interface exists for that particular direction.

4.1 Communication Layer

Intuitive description:

The communication layer provides a simple and clear, protocol agnostic, message passing interface to the layers above. This layer the details involved in the network protocols (such as TCP, UDP and IP) used for the actual message transfer. The functionality of the layer resembles that of a post office. It supports two services: a reliable message delivery, and an unreliable best-effort one. Neither service guarantees FIFO delivery. The design of this layer assumes fair communication links, as defined in section 3.2.

Interfaces:

MessageTransmitter:

The MessageTransmitter interface, though which the Communication layer interacts with layers above it, utilizes two concepts: the concept of Outbox, and the concept of StrongOutbox. Each one of these two concepts is a local data structure for each process where other layers of this process can add messages.

- StrongOutbox: Set of Messages

If both the sender (source) and the receiver (destination) of a message added to the StrongOutbox are correct processes then the message is guaranteed to be delivered to the destination process. A layer can *add* a message to the StrongOutbox data structure by calling a method called addToStrongOutbox(m: Message).

- Outbox: Set of Messages

Outbox is a weaker version of StrongOutbox. It only guarantees best-effort delivery of messages exchanged between correct processes. A layer can *add* a message to the Outbox data structure by calling a method called addToOutbox(m: Message).

MessageConsumer (Call Back Interface):

Whenever a message is received from the real network, the Communication Layer is responsible for passing this message to the appropriate layer. To achieve this, each layer that needs to receive messages must implement the MessageConsumer interface. Note that messages store information indicating the layer they belong.

- consumeMessage(m: Message): void

Properties:

We say that a layer L of process p *delivers* a message m, if the Communication layer of p calls the consumeMessage method of layer L with m as an argument. In this case, we also

say that p *delivers* m. Message m stores information about its source and destination processes.

Formally Outbox satisfies the following safety property:

- Validity: If some layer L of a process p_i delivers a message m, and the sender of m is a process p_j, then layer L of p_j has previously added m to its Outbox, StrongOutbox, or both.

StrongOutbox satisfies the above validity property along with the following liveness property:

- Liveness: If some layer L of a correct process p_i adds a message m to its StongOutbox and m's destination is a correct process p_j then layer L of p_j eventually delivers m.

Implementation Overview:

We use TCP/IP connections for inter-process communication. The Communication layer implementation of a process periodically examines the contents of its Outbox and StrongOutbox and attempts to send each message to its destination process. A message is removed from the Outbox regardless of whether the attempt is successful, while only messages that are successfully transmitted to their destination are removed from the StrongOutbox. The Validity property described above follows from the TCP/IP connection guarantees. The Liveness property follows from the fairness of

communication channels and the fact that we only remove from the StrongOutbox messages that have been successfully delivered to their destination.

4.2 Failure Detector Layer

Intuitive description:

The purpose of this layer is to provide the layers above with an educated opinion about the status of the other servers in the system. For every process it maintains a *health level*, namely an integer value between minHealthLevel and maxHealthLevel values. The higher a health level of a process, the more likely it is that this process is still alive.

Interfaces:

FailureDetector

- getHealth(p: ProcessID): Integer

This method returns the health level of process p.

MessageConsumer

This is the interface called by the Communication layer whenever it receives a message for the Failure Detector layer.

MessageTransmitter (Call Back Interface)

This is the interface used by the Failure Detector layer in order to broadcast "I am alive" messages. It is implemented by the Communication layer.

Properties:

A Process p regards p′ as crashed if return value h of an invocation of the method getHelath(p′) is less than a zeroHealthLevel, that is between minHealthLevel[3] and maxHealthLevel. In this case we say that p *suspects* p′. The process p *permanently suspects* p′ if there is a point in time after which any invocation of getHelath(p′) by p returns a value that is less than zeroHealthLevel.

We assume that the Failure Detector belongs to class ◊S [1], namely it satisfies the following two properties:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected by correct processes.

Implementation Overview:

Every T_{FD} time units, the failure detector of each process broadcasts an "I am alive" message to all the servers in the system, using the Outbox. The health level h that a process p keeps for a process p′ is updated as follows. The value of h is decreased by 1 every T_{FD} seconds, unless h is equal to minHealthLevel, or p has received an "I am alive"

[3] minHealthLevel is needed in the dynamic case.

message from the failure detector layer of p′ within the last period. In the latter case, h is set to maxHealthLevel. Strong Completeness follows from the use of a finite T_{FD} and the Validity property of the Communication layer. Finally, we assume that in practice there exists a finite T_{FD} long enough for which the TCP/IP protocol used to implement the Outbox guarantees the Accuracy property.

4.3 Core Consensus Layer

Intuitive Description:

The Core Consensus Layer of each server process p maintains a set R_p of request IDs. The Core Consensus layer is responsible for determining a global order on the IDs of *stable* requests, i.e., requests that appear in the R-sets of the majority of the servers. This is achieved by repeatedly executing a t-resilient distributed consensus protocol among the servers. Request IDs are added to R_p, asynchronously by the Upper Consensus layer of p, which is the unique layer on top of the Core Consensus layer of p. The Core Consensus layer of p progressively generates an ordered list L_p of request IDs, that grows by appending new elements to the end of L_p. Every execution of the consensus protocol appends one or more request IDs to L_p. It is guaranteed that the L-lists of all processes are prefixes of the same infinite sequence L_∞ of request IDs, that all stable requests, and that only them, appear in L_∞, and each appears exactly once.

Interfaces:

Consensus

- addProposal(s: Proposal): Void

This method is called by the Upper Consensus layer to add a request ID s to the local R set.

- nextConsensus (): List of request IDs

The invocation of this method triggers the next execution of the consensus protocol. Invocations of the nextConsensus method take place asynchronously in different processes. The return value is a non-empty list of request IDs. We require that the layer above that invokes the method waits until a reply to the last invocation is received before it makes a new call to this method.

MessageConsumer

This is the interface called by the Communication layer whenever it receives a message for the Core Consensus layer.

MessageTransmitter (Call Back Interface)

This is the Communication layer interface used by the Core Consensus layer in order to communicate with Core Consensus layers of other processes.

FailureDetector (Call Back Interface)

This is the interface implemented by the Failure Detector layer that allows the Core Consensus layer to retrieve information about the correctness of other processes.

Properties:

Every execution of the nextConsensus method by some process p marks a different *core consensus cycle*[4], or simply *c-cycle*, for p. In particular, the i-th c-cycle of p begins when the execution of the (i-1)-th call of nextConsensus by p returns, and we say that p *enters* c-cycle i. The i-th c-cycle of p ends upon the termination of the execution of the i-th call of nextConsensus method of p, and we say that p *completes* c-cycle i. Moreover, we say that p is *in* c-cycle i from the moment p enters c-cycle i until p completes c-cycle i. For every request ID s in the list λ returned by the i-th execution of nextConsensus of a process p, we say that p *c-decide*[5]*s* s with *order*[6] r in c-cycle i, where r is the order of s in λ. We also say that p *c-decides a list* $< s_1, s_2, \ldots, s_\kappa >$ in c-cycle i, if p c-decides s_j with order j in c-cycle i, for j=1, 2,…, κ. Finally, we say that a process p *c-decides* in c-cycle i if there exists a proposal s such that p c-decides s with order r in cycle i, for some r.

We require that the Core Consensus layer satisfies the following properties:

- Validity 1: If a process p c-decides a request ID s (with some order in some c-cycle) then a majority of processes have previously executed addProposal(s) of Core Consensus layer.
- Validity 2: A process does not c-decide the same request ID more than once in the same or different c-cycles.

[4] Along the same line of the footnote 3, a better name is simply "consensus cycle".
[5] Along the same line of the footnote 3, a more appropriate name for "c-decides" is simply "decides".
[6] A more appropriate name for the term "order" in this context is "rank".

- Uniform Agreement: For any processes p, p′ and c-cycle i, if p c-decides some list of request IDs *lst* and p′ c-decides some list of request IDs *lst′* in cycle i then *lst* = *lst′*.
- Liveness 1: If all correct processes execute addProposal(s), and every correct process p eventually calls nextConsensus again after the previous invocation of nextConsensus by p returns, then eventually some correct process q c-decides s (in some c-cycle).
- Liveness 2: A If a process p completes its i-th c-cycle and a correct process q enters its i-th c-cycle then q completes its i-th c-cycle.

Implementation Overview:

Our core consensus algorithm is a rotating coordinator based algorithm which proceeds in asynchronous rounds. The algorithm we implemented is the consensus sub-algorithm of the uniform atomic broadcast protocol (with linear message complexity) described in [10] and is briefly overviewed below.

A cycle i of the Core Consensus layer of a process practically begins when the nextConsensus method is called for the i-th time. The algorithm proceeds in asynchronous rounds. In each round the Core Consensus layer of some server serves as a *coordinator* that tries to impose its estimate as the decision value of Core Consensus layers of all server processes for this cycle. For wording simplicity in the remaining of this sub-section we refer to the Core Consensus layer of the coordinator process simply as

the *coordinator*. Similarly we refer to the Core Consensus layers of all server processes simply as *participants*. Note that the coordinator is also a participant. We also refer to the Failure Detector layer as the failure detector. Furthermore, all messages transmissions are done through the strong outbox facility of the Communication Layer, which guarantees reliable message transmission.

For load balancing, the coordinator of the first round of each cycle is different than the coordinator of the first round of the previous cycle, determined by an arbitrary cyclic ordering of processes O. In fact, the coordinator is uniquely determined by the cycle and round number. Each participant periodically queries its failure detector about the health level of the coordinator. If the failure detector reports the coordinator as suspected then the participant expects that some new coordinator will start a new round. In particular, for each participant the next coordinator is expected to be the first process in the cyclic ordering O that is not suspected. If a process expects that it is the next coordinator, then it stops any of its activities regarding the current round and proceeds as the coordinator of the next round that corresponds to it. The asynchrony of the rounds means that processes do not need to synchronize when changing rounds and thus at the same time different processes may be in different rounds. If any participant p receives a message that belongs to a round r than is greater than the current round r′ of p, then p abandons any activities regarding round r′ and starts participating in round r. Moreover, if any participant receives a message that belongs to a higher cycle, it abandons the current cycle and broadcasts a special decision-request to all processes asking for the decision list of all the cycles it missed.

Each round consists of 3 stages for the coordinator and 3 stages for each participant. Recall that the coordinator is a participant too, so it executes the 3 stages of the participants as a parallel task. To ensure uniform agreement of decisions, each process *adopts* a decision list as a local estimate before it c-decides this estimate. The coordinator of each round has to take into consideration any estimates possibly broadcast by coordinators of previous rounds of the same consensus cycle. For this reason, before the coordinator adopts an estimate, all participants inform the coordinator about any estimates they have previously received and adopted. Figure 10 depicts the stages that the algorithm follows in a round. In more detail, the stages are as follows.

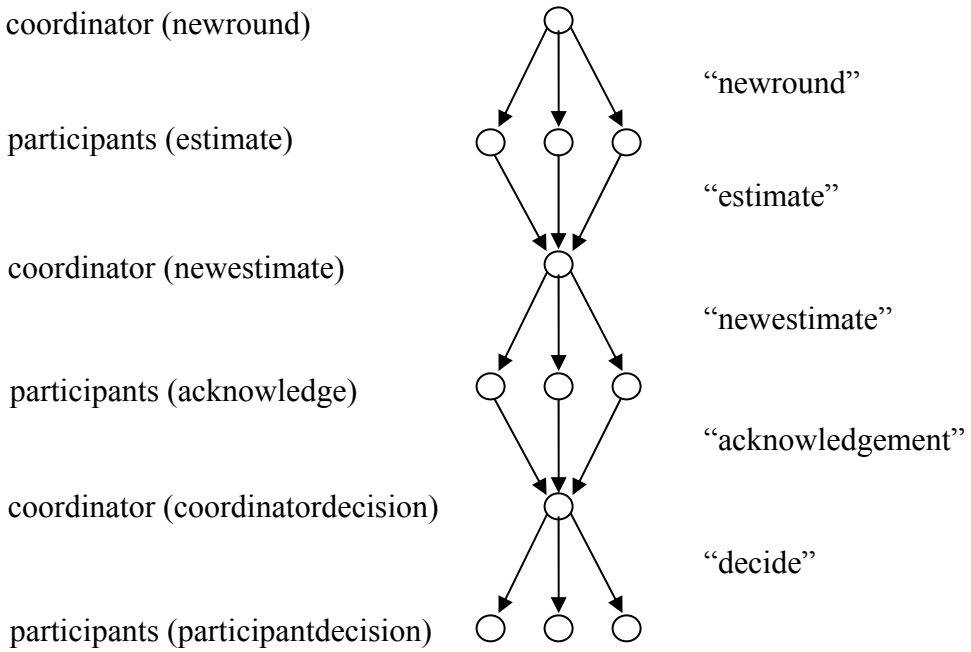

Figure 10 Stages that the algorithm follows in order to take a decision. Circles represent processes, arrows represent messages. An arrow starting from a process p and pointing to a process q, means that p sends a message to q. The label to the left of a circle indicates the type of the corresponding process and its current stage, whereas the label to the right of an arrow indicates the type of the corresponding message.

In the coordinator’s stage 1 (namely: *newround* stage) the coordinator sends a “newround” message to all participants and proceeds to its stage 2 (namely: *newestimate* stage).

When a participant receives a “newround” message it stops any activities of any previous rounds and starts participating in the current round by entering a participant’s stage 1

(namely: *estimate* stage). In the estimate stage all participants send any estimate they have adopted so far to the coordinator. They also periodically send to the coordinator all the request ids they have received through their addProposal methods and have not been c-decided yet in any previous cycle. Each participant stays in this stage until it has received a "newestimate" message from the coordinator (or suspects the coordinator).

The coordinator in its "newestimate" stage waits until it has received the estimates of at least t+1 distinct participants. When this is accomplished it needs to adopt a non-empty list of request ids as it estimate. This adoption can happen in two ways. If there exist "non-null" estimates it means that a previous coordinator has already proposed a decision list for this cycle and some processes may have already c-decided this list. In this case, the coordinator adopts this estimate as its estimate. Otherwise, if all the "estimate" messages that the coordinator received are "null" then it waits until there exists a non-empty set of request ids proposed by at least t+1 distinct participants. It then arbitrarily orders this set and adopts this ordered list of request ids as its estimate. When the coordinator has adopted an estimate, it sends a "newestimate" message to all participants, which includes its estimate, and proceeds to its stage 3 (namely: *coordinatordecision* stage)

When a participant receives a "newestimate" message it proceeds to its stage 2 (namely: *acknowledge* stage). In this stage it simply adopts the estimate received from the coordinator, it sends an "acknowledgement" message to the coordinator, and it proceeds to its stage 3 (namely: *participantdecision* stage).

The coordinator process stays in its stage 3 until it has received "acknowledgement" messages from at least t+1 distinct participants. When this requirement is fulfilled, it sends a "decide" message to all participants with its estimate.

Each participant process stays in its stage 3 until it has received a "decide" message from the coordinator. When a "decide" message is received the list contained in that message is considered as the decision list of this cycle and the nextConsensus method returns this list.

The properties of this layer can be derived directly from the corresponding properties proven in [10], and the guarantees of our Communication and Failure Detector layers. In more detail, Validity 1 is simply a restatement of Lemma 3. Validity 2 follows from the (stronger) result described in Theorem 18. Uniform Agreement follows from Corollary 9, and the fact that request IDs are unique, so we can deterministically order the members of any set of request IDs, e.g., in their lexicographic order. Liveness 1 is a consequence of Lemma 12 and Lemma 15. Finally, Liveness 2 follows directly from Lemma 14.

4.4 Upper Consensus Layer

Intuitive Description

The Upper Consensus layer is responsible for uniformly atomically broadcasting to all servers every client request that is received by any server. In more detail, every time a server p receives a client request r, it passes r to its Upper Consensus layer. Then, the Upper Consensus layer of p broadcasts r to the Upper Consensus layers of all servers. Every Upper Consensus layer that receives r passes the ID of r to its Core Consensus layer, which is responsible for fixing a global order on the broadcasted request IDs. The layer above the Upper Consensus layer (Server Management layer) periodically asks the Upper Consensus layer for new requests to process. The Upper Consensus layer guarantees that, in every server, the same client requests are returned in exactly the same order. Moreover, every client request will be returned, eventually.

Interfaces

Consensus

- addProposal(s: Proposal): Void [7]

  This method is called by the Management layer to pass a client request, s, to the Upper Consensus layer. s is then broadcasted by the Upper Consensus layer of this process to the Upper Consensus layers of all the servers.

- nextConsensus (): List of request IDs [8]

  The invocation of this method simply calls the nextConsensus method of the Core Consensus layer and returns the list of requests that corresponds to the list of request

[7] Along the same line of the footnote 2, a more appropriate name for this method is “addMessage”
[8] Along the same line of the footnote 2, a more appropriate name for this method is “nextList”

IDs returned by the nextConsensus method of the Core Consensus layer. We require that the layer above that invokes the interface waits until a reply to the last invocation is received before it makes a new call to this method.

MessageConsumer

This is the interface called by the Communication layer whenever it receives a message m for the Core Consensus layer. If m is a broadcasted request then the consumeMessage method calls the addProposal method of the Core Consensus layer with m's ID as an argument.

MessageTransmitter (Call Back Interface)

This is the Communication layer interface used by the Upper Consensus layer in order to communicate with Upper Consensus layers of other processes.

Properties

We define u-cycles with respect to invocations of the nextConsensus method of the Upper Consensus layer in exactly the same way as we defined c-cycles for the Core Consensus layer. In fact, the i-th u-cycle of a process p coincides with the i-th c-cycle of p. Thus, we can refer to both of them as *cycles*. We define the term u-decide[9] similarly to

[9] Along the same line of the footnote 2, a more appropriate name for "u-decide" is "deliver"

the term c-decide we defined in Section 4.3. We also use the phrases “a process P enters/completes a u-cycle”, or “p is in a u-cycle” in the expected way.

We require that the Upper Consensus layer satisfies the following properties:

- Validity 1: If a process p u-decides a request s (with some order in some u-cycle) then some process have previously executed addProposal(s) of Upper Consensus layer.
- Validity 2: A process does not u-decide the same request more than once in the same or different u-cycles.
- Uniform Agreement: For any processes p, p′ and u-cycle i, if p u-decides some list of requests *lst* and p′ u-decides some list of requests *lst′* in cycle i then *lst* = *lst′*.
- Liveness 1: If a correct process execute addProposal(s), and every correct process p eventually calls nextConsensus again after the previous invocation of nextConsensus by p returns, then eventually some correct process q u-decides s (in some u-cycle).
- Liveness 2: If a process p completes its i-th u-cycle and a correct process q enters its i-th u-cycle then q completes its i-th cycle.

Implementation Overview:

The Upper Consensus layer of a process broadcasts the requests passed to it from the Server Management layer to the Upper Consensus layers of all servers using the

StrongOutbox. The Upper Consensus layer of every server that receives the request stores the request locally and passes its ID to the Core Consensus layer using the addProposal method. Every time the Server Management layer calls the nextConsensus method, the Upper Consensus layer invokes the nextConsensus method of the core consensus layer and returns the list of requests that corresponds to the list of request ID returned by the Core Consensus layer. If a server cannot reconstruct a request by using locally stored information then it broadcasts an ID resolution message to all servers. At least one correct server must have the required information; this follows from the Validity 1 property of Core Consensus.

The Validity 1 property of Upper Consensus follows from the Validity 1 property of Core Consensus and the fact that a message broadcast by a correct server is received by every correct server. The remaining properties follow directly the corresponding properties of Core Consensus layer and the fact that each request has a unique ID.

4.5 Management Layer

The role of the Management layer is to reliably and transparently connect the Client Application layer and the Server Application layer utilizing the layers below the Management layer. By “reliably” we mean that the system can tolerate server crashes, while “transparently” means that the Management layer hides server replication and all

the issues related to it from the Server and Client Application modules, as we showed in Figure 1b.

In more detail, the Management layer is responsible for replicating the Server Application module across the server processes, and for delivering every client request to all server application modules in the same order. It is also responsible for returning a reply for every request to the client application module that generated the request. It assumes that the server application module is deterministic. So, it runs several copies of the server application module concurrently on the servers it utilizes, it provides all of the server applications with the requests generated by the client applications in the same order and returns the replies of the server applications to the client applications. Client applications believe that they are connected to a single server application and each server application believes that it is the only server in the system.

Part of the Management layer is implemented in the client side and part of it in the server side.

4.5.1 Server Management layer

Intuitive Description:

The Server Management layer has limited responsibilities in the static server model. For every request r it receives from the Management layer of some client, the Server Management layer should respond to client with the appropriate reply q, in the sense that

q is the reply that the Server Application module would generate if all the requests sent by all clients were serialized and processed by a single Server Application module. To accomplish this, the Server Management layer passes to the Upper Consensus layer every request it receives from the Management layer of a client. Periodically, it asks the Upper Consensus layer to provide a new request r, which is passed to the Server Application layer for processing. If r is a request that was originally sent to the current server by some client c then the generated reply is sent back to the Management layer of c.

Interfaces:

MessageConsumer

This interface of the Server Management layer is called by the Communication layer whenever it receives a message m for the Server Management layer.

MessageTransmitter (Call Back Interface)

This is the Communication layer interface used by the Server Management layer in order to communicate with the Management layer of server or client processes.

Consensus (Call Back Interface)

This is the Upper Consensus layer interface used by the Server Management layer to pass requests to the Upper Consensus layer for ordering and to retrieve ordered requests for processing.

RequestApplier (Call Back Interface)

This is the interface that the Server Application layer implements to allow the Server Management layer to pass client requests for processing. It consists of a singe method:

- applyRequest(r: Request): Reply

FailureDetector (Call Back Interface)

This is the interface implemented by the Failure Detector layer that allows the Server Management layer to retrieve information about the correctness of other processes.

Application

This interface is used by the Server User Interface layer to pass to the Server Management layer control commands issued by the system administrator. Typically, the system administrator uses the Server User Interface to issue to the Server Management layer commands regarding the operation of the server, e.g., start/stop the server, or simulate message delays and process crashes for testing purposes.

We will describe this interface in more detail in Section 4.7.

UI (Call Back Interface)

The UI call back of Server User Interface layer is used by the Server Management layer to respond to commands issued by the system administrator, and output monitoring information. Again, we will talk about the UI interface more thoroughly in Section 4.7.

Properties:

- Liveness: Let p be a correct process such that the Server Application layer returns a reply for every request passed to it by the Server Management layer. The Server Management layer of p eventually sends a response to the Client Management layer of any client for every request it receives from the Client Management layer of that client.
- Integrity: If the same client request is received by the Management layer of more than one server, or by the Management layer of the same server more than once, then the same reply is generated in all cases.
- Linearizability: For every execution of the system, there exists a total ordering T of all client requests that are received by the Management layers of servers and for which there exists a matching reply, such that every reply that is generated for each of these requests is the same as the one that a single server would produce if the same set of requests were submitted to this server sequentially in order T. Moreover, T respects the actual order of non-overlapping request-reply pairs.

Implementation Overview:

The Management layer of a server s executes two tasks in parallel. The first task receives requests from the corresponding layer of clients. If the layer already has a reply for some received request stored in a locally maintained set Q_s, then it sends the reply back to the client. Otherwise, if the request is not already stored in a locally maintained list of

"pending requests" PR_s, it is passed down to the Upper Consensus layer and is also stored in PR_s. The second task invokes the nextConsensus method of the Upper Consensus layer sometime after the previous invocation of this method is completed. The same method is also invoked once during the initialization of s. The list of requests returned by every invocation of nextConsensus method is passed up to the Server Application layer, and the replies generated by the Application layer are stored in Q_s. Finally, if any request r in PR_s corresponds to some of these replies then the appropriate reply is send back to the source client of r.

The Liveness property follows by the following argument. Let r be a request that is received by the Management layer of a correct server s. If a reply for r already exists in Q_s, then our protocol immediately sends a reply back to the client. Otherwise, r is passed to the Upper Consensus layer. Since the Management layer invokes the nextConsensus method of the Upper Consensus layer immediately after the previous invocation of this method is completed, the Liveness 1 and 2 properties of Upper Consensus imply that the Management layer of s will eventually get back a list that contains r. Thus, r is passed to the Application layer. By the hypothesis of the Liveness property of the Server Application layer, the Server Application layer will return a reply q for r. Since we assume that s is correct, our implementation ensures that q is sent back to the client that issued r.

From the protocol of the Server Management layer it is clear that the Integrity property holds if exactly the same sequence of requests is provided by the Upper Consensus layer

of every server, and this sequence contains no duplicates. These two conditions are guaranteed by the Uniform Agreement and Validity 2 properties of Upper Consensus layer.

The Linearizability property follows by the same argument as the Integrity property and the additional requirement that the sequence of requests contains only valid entries, i.e., requests that have been received by the Management layer of some server. This requirement is precisely the Validity 1 property of Upper Consensus layer.

4.5.2 Client Management Layer

Intuitive Description

The Client Application module views the management layer on the client side as a server application module. Thus, the Client Management layer has to implement the same interface as the modules of the Server Application layer. To maintain an application independent design, we require that Client and Server Application modules communicate via a standard generic interface, called RequestApplier. In fact, the Client Management layer simply communicates the Client requests to the servers and returns the replies back to the Client Application layer.

Interfaces

RequestApplier

- applyRequest(r: Request): Reply

For simplicity, we require that the Application layer in a client process waits until a reply to its last request to the Client Management layer is received before it passes a new request.

MessageConsumer

This is the interface called by the Communication layer whenever it receives a message m for the Client Management layer.

MessageTransmitter (Call Back Interface)

This is the Communication layer interface used by the Client Management layer in order to communicate with the Server Management layers of server processes.

Properties:

Assuming that clients are correct and that the Server Application layer generates a deterministic reply to every client request, we require that the Client Management layer satisfies the following properties:

- Liveness: Every call of applyRequest returns a Reply.

- Linearizability: For every execution I_{ms} of the system, there is an execution I_{ss} in the single server model (with the same set of clients and the same set of client requests), which is indistinguishable from I_{ms} from any client's point of view. In other words, there is a serialization of the request-reply pairs in I_{ms} that respects the ordering of non-overlapping pairs, and the reply to each request is the same as the reply a correct server would generate, if the requests that have a matching response were sent to the server sequentially in the serialization order.

Implementation Overview:

The Client Management layer of a client c sends the requests that the Client Application layer passes to it to the Management layer of some server s. We choose s to be the server that responded to the most recent request issued by c (or a randomly chosen server, if this is c's first request). If a reply to that request is received within a predefined timeout, the reply is passed up to the Client Application layer. Otherwise, the same request is sent to a randomly selected server among the remaining servers. The same procedure is repeated until a reply is received. A reply may arrive from a server other than the one that the request was originally sent to by the client.

This approach satisfies the Liveness property because of the Liveness property of the Server Management layer, the Liveness property of the Server Application layer, the Liveness property of the Communication layer, and the fact that the Management layer of the client repeatedly sends requests to t+1 servers.

The Linearizability property follows from the Linearizability property of the Server Management layer, and the Integrity property of the Server Management layer.

4.6 Application Layer

This layer implements the application-specific part of the system. This layer is present in both client and server processes. Typically, a server Application layer consists of more than one Server Application modules, whereas the client usually runs a single Client Application module.

4.6.1 Server Application Layer

Intuitive Description:

This layer can host several server applications as different modules. Each application does not know about the other applications, and does not interfere with them.

Interfaces:

RequestApplier

- applyRequest(r: Request): Reply

This method is invoked by the Server Management layer when a request is available for processing. We require that a new invocation to the applyRequest method takes place only after any previous invocation has returned.

Properties:

- Liveness: Any call to the applyRequest method of the Server Application layer of a correct process returns a Reply.
- Determinism: If two copies of a Server Application module receive the same sequence of requests they produce the same sequence of replies.

The Liveness property does not restrict the generality of the system, while the Determinism property hold for a wide range of applications.

Implementation Overview:

We built a simple application that maintains a shared integer register. It supports a read operation, a write operation, and the four basic integer arithmetic operations (+, -, ·, ÷). The first one returns the current value of the register, while the others return an "ACK".

4.6.2 Client Application

Intuitive Description:

The purpose of this layer is to decode the actions of the user received from the User Interface layer and to form requests for these actions. The requests are then passed to the Client Management layer which returns a reply for each of these requests. Finally, the Client Application layer informs the User Interface accordingly.

Interfaces:

Application

This interface is called by the Client User Interface layer to pass user commands. We will describe this interface in detail in Section 4.7.

UI (Call Back Interface)

This interface is implemented by the User Interface layer. Its purpose is to inform the user about the state of the system. See Section 4.7 for details.

RequestApplier (Call Back Interface)

This is the interface the Client Management layer should implement to allow the Client Application layer to pass client requests for processing.

Properties:

- This layer handles user commands properly, i.e., by constructing the appropriate requests and passing them to the attached RequestApplier interface. When it receives a reply it informs the user by updating the User Interface appropriately.

Implementation Overview:

The client part of our application that maintains a shared integer register translates user actions into requests, passes them to the Client Management layer, receives the replies, and updates the User Interface.

4.7 User Interface Layer

Intuitive Description:

The User Interface layer typically connects client users with the Client Application layer and system administrator users with the Server Management layer. We call both client and system administrator users as *users*. In more detail, the User Interface layer is used by users to issue system commands, and by the system to output information to users. It can be implemented as a GUI, command line, pipes, library calls, or even files on a hard disk.

The Server User Interface could interact with the administrator of the system, or it can just log the decisions taken during the execution to a file.

The User Interface is assumed to have some input/output functionality. The input received by the user is converted into a list of strings and passed to the layer below through the Application interface. The interpretation of each string is implementation-specific. For instance a string can be associated with the value of a text box, or the id of a button pressed, etc. The output functionality, implemented via the UI interface, must support a mechanism to display text, such as through a graphical text box, the console or a file. We call this mechanism a *screen*.

Interfaces:

UI[10]

- appendString(s: String): void

This method appends s to the screen.

- clearScreen(): void

This method clears the screen.

Application[11] (Call Back Interface)

- newCommand(s: Array of Strings): void

The content and representation of s by the User Interface layer depends on the implementation. It does not return anything but may trigger the invocation of one or more of the methods of UI to make the results of this command visible to the user.

Implementation Overview:

We implemented a simple GUI that allows the client user to send commands to the integer register manually, or offers the choice of automatic submission of a randomly generated sequence of commands. We also built a simple server GUI that displays server monitoring information.

[10] A more descriptive name for this interface is UIFront
[11] A more descriptive name for this interface is UIBack

Chapter 5

The Dynamic Server/Agent Set Version

To be able to substitute crashed servers with correct ones we introduce the concept of agents. Agents are passive backup servers that are standing by, waiting to become fully active servers when they are asked to do so. Until then, they have limited responsibilities compared to the actual servers, although they have exactly the same structure and functionality as the servers. The replacement of crashed servers by agents is basically achieved by employing a special Server Application module, called Server Group Membership (SGM), and by extending the functionality of the Server Management layer. Some minor modifications are also applied to other layers of the system, as well. The idea of using an atomic broadcast algorithm that orders requests also to maintain the set of current servers dynamically first appears in [13].

In more detail, the Server Management layer of a server p periodically retrieves from the Failure Detector layer the list of health levels of all processes (servers and agents). If some server s is seriously suspected by p, i.e., the health level of s at p is equal to minHealthLevel, then the Management layer of p generates a *suspicion report* r request and passes this request to the Upper Consensus layer, where it is treated as a regular application request. Eventually, r will be returned by some invocation of the nextConsensus method of the Upper Consensus layer. Whenever the Management layer of a process q receives a suspicion report request by the Upper Consensus layer it passes the request to the SGM module. Based on the suspicion requests the SGM module has received so far, it decides whether a replacement in the set of servers should take place, and informs the Management layer accordingly by returning a *server replacement* order. When the Management layer of q receives a server replacement order indicating that a server s_{old} should be replaced by some agent s_{new}, q sends to s_{new} a copy of the full state of q and informs the Upper Consensus layer that the set of servers have changed.

A change in the set of processes that p regards as servers marks the beginning of a new *epoch* E of p, and the end of the previous epoch (E-1) of p. At the moment when the Management layer has been informed of the change by the SGM module we say that p *enters* epoch E. Although epoch changes occur asynchronously in different processes they are globally consistent, in the sense that epoch E spans the same sequence of cycles in every process. This is guaranteed by the fact that suspicion requests arrive at the SGM module of every process in the same order. In addition, the moment that at least t+1 processes enter an epoch E we say that the *system enters epoch E*. For the duration of

time that passes from the moment the system enters an epoch E until the moment the system enters epoch E+1, we say that the *system is in epoch E*. We require that at most t servers of any epoch E may crash during the time the system is in epoch E. Moreover, we require that at least t+1 servers of any epoch E are also servers of epoch E+1. These two requirements guarantee that for any two consecutive epochs there exists at least one server that is in both of them and is correct during these epochs. Figure 11 depicts a portion of a system execution consisting of ten cycles that are divided in three epochs.

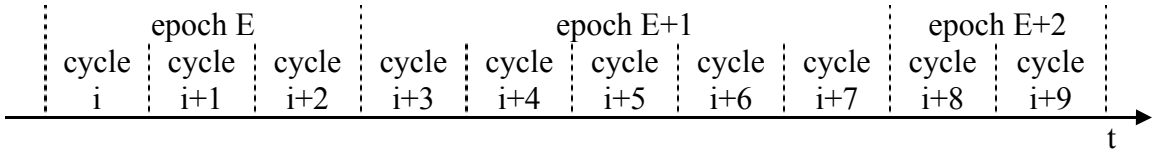

Figure 11 A portion of a system execution consisting of ten cycles that are divided in three epochs.

So far we have roughly described how changes to the set of servers take place, but we have not argued for the use of such changes. Ideally, every time an active server crashes we would like to replace it by a correct agent. This requirement, however, is not feasible if only a ◊S failure detector is used, since with such a Failure Detector we cannot conclude with certainty whether a process has crashed or is just slow. Instead, we adopt an approach that does not support such strong guarantees, but we believe that in practice it achieves similar results. This approach is described in detail in Sections 5.2.5 and 5.2.4.1, and works roughly as follows. The SGM module of a process p orders the replacement of a server s_{old} by a server s_{new} only if s_{old} is seriously suspected by a large number of servers and s_{new} is regarded to be in good health state by a large number of servers. A more general view of this approach is that changes in the set of servers aim at

improving the *health level of the system*, which is defined in terms of the health levels that every server maintains about the other servers in the system.

5.1 Overall Architecture and the Snapshot Interface

The architecture of client processes remains the same as in the static case, while the design of servers changes slightly to accommodate the ability to dynamically change the set of servers. Both servers and agents are equipped with exactly the same layers. They only differ in that some layers in agents are not active. Figure 12 shows the layering of a server/agent. The shaded boxes correspond to layers that are not active in an agent.

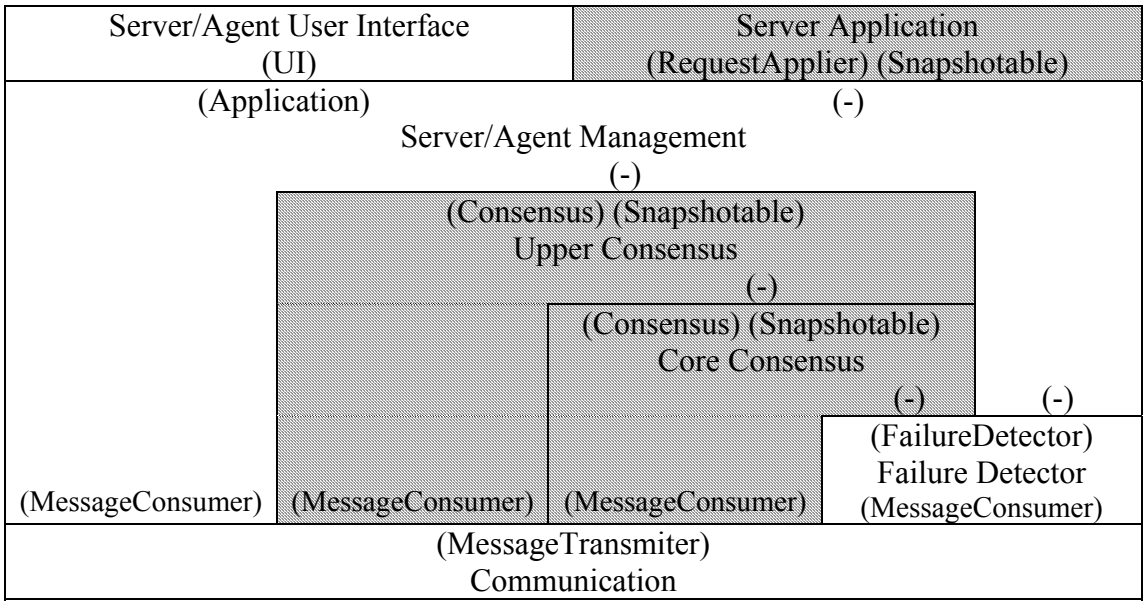

Figure 12 Layers of servers/agents. We use the same convention as in Figure 9. In addition, the shaded boxes are layers that are deactivated in agents, and become active when an agent becomes a server.

To enable a server to send a copy of its state to an agent, we extend the specification of its Core Consensus, Upper Consensus, and Application layer by adding a snapshot functionality, which returns an exact copy of the internal state of the corresponding layer. More precisely, each of these layers implements the following additional interface:

Snapshotable

- getState(): Snapshot

  This method causes the layer to temporarily pause its operation and to produce an object of type Snapshot. The information stored in this object is enough to accurately reproduce the state of the layer.

- setState(s: Snapshot): void

This method overwrites the layer's internal state with the state described in s.

Both methods should not be invoked concurrently with any method of any interface of a layer in the same process.

Specific changes applied to each layer are described in the next section.

5.2 Specific Layer Modifications

It is worth noting that the Liveness property of all layers of the static case, with the exception of the Liveness property of the Client Management layer, has to be modified since they all depend on the definition of a correct process. A correct process in the dynamic case is defined only within the bounds of a specific epoch. Thus, the Liveness properties can be satisfied only if the execution contains a last infinite epoch. We modify the Liveness properties of each layer accordingly to comply with the above statement.

The Liveness properties we define for the dynamic case might seem too weak to be useful. However, this is not the case, since the execution of the system in the dynamic case can be considered as the concatenation of finite prefixes of infinite executions of the system in the static case. In particular, for every execution I of the system in the dynamic case there exists an infinite execution I_∞ of the system in the dynamic case such that before a process enters epoch E+1 in I, both executions are indistinguishable. Liveness guarantees that in I_∞ every request issued during epoch E will generate a reply in a finite amount of time. Thus, all replies generated in I_∞ at any time before any process enters

epoch E+1 in I will also be generated in I. So, if in I epoch E lasts long enough some requests will generate a reply. Under the assumption that only a bounded number of epoch changes may occur, there is always an epoch E that satisfies the liveness property, namely the last one. Moreover, if every client request that has not received a reply is resent to some correct server during this last epoch, then the stronger liveness property of the Server Management layer in the static case holds in this case as well.

5.2.1 Communication Layer

The Communication layer in the dynamic case can satisfy the Liveness property only if there exists a last infinite epoch:

- Liveness: If there is a last infinite epoch E and some layer L of a correct process (of E) p adds a message m to its StongOutbox and m's destination is a correct process (of E) q then layer L of q eventually delivers m.

5.2.2 Core Consensus Layer

Since the set of servers is dynamic, both consensus layers need to know the exact set of processes that participate in every cycle. The set of servers a process p considers to be participating in an execution of the nextConsensus method of p remains fixed during this execution; it can only change after the termination of an execution of nextConsensus and

before the next invocation by p. The set of servers changes by invoking the nextEpoch method, which extends the Consensus interface of the Core Consensus layer of static servers. An invocation of nextEpoch of p marks the beginning of a new epoch of p.

Interfaces:

The Core Consensus layer in the dynamic case differs from that it the static case in two ways. First, the Snapshotable interface described in 5.1 is added to allow capturing and setting the internal state of the layer. Second, the Consensus interface is extended by the addition of the following method that handles epoch changes:

- nextEpoch (participants: Set): Void

This method must not be invoked concurrently with nextConsensus method or other invocations of nextEpoch by the same process.

Properties:

We require that the following preconditions hold:

- Precondition 1: If a process calls nextEpoch(S) at the beginning of some cycle c, for some set of servers S, then every process that calls nextConsensus during c must have previously called nextEpoch(S) at the beginning of c.
- Precondition 2: Let S be the set of servers of some epoch E, and S´ be the set of servers of epoch E-1. If s is in S-S´ then s enters epoch E only after its Core Consensus layer has been initialized with the state that one of the remaining processes p of S has immediately before p enters epoch E, and the R-sets of all servers are emptied at the beginning of epoch E.

Given the above preconditions, the same set of properties holds as in the static case, with the exception that Liveness 1 is slightly modified as follows:

- Liveness 1: If there is a last infinite epoch E, all correct processes (of E) execute addProposal(s) during E, and every correct process (of E) p eventually calls nextConsensus again after the previous invocation of nextConsensus by p returns then, eventually, some correct process (of E) q c-decides s.
- Liveness 2: If there is a last infinite epoch E and a process p completes its i-th c-cycle during E then if a correct process (of E) q enters its i-th c-cycle then q completes its i-th c-cycle.

Implementation Overview:

The getState method of this layer resets the R set of the layer before it captures and returns the internal state of this layer. The nextEpoch method simply changes the set of processes the Core Consensus layer regards as servers.

It is not possible that the set of server changes in the middle of the execution of a consensus, because nextEpoch is not executed concurrently with any execution of nextConsensus by the same process. From Preconditions 1 and 2, and the inductive hypothesis that the properties of static case hold until the last cycle of epoch E-1 it follows that the same properties still hold in epoch E, since at the beginning of epoch E all correct servers have exactly the same history and empty R-sets.

5.2.3 Upper Consensus Layer

The Upper Consensus layer of the dynamic case differs from the one of the static case in a way similar to how the Core Consensus layer of the dynamic case differs from the Core Consensus layer of the static case.

Interfaces:

The Upper Consensus layer in the dynamic case differs from that it the static case in two ways. First, the Snapshotable interface described in 5.1 is added to allow capturing and setting the internal state of the layer. Second, the Consensus interface is extended by the addition of the following method that handles epoch changes:

- nextEpoch (participants: Set): Void

This method should not be invoked concurrently with nextConsensus method or other invocations of nextEpoch by the same process.

Properties:

We require that the following preconditions hold:

- Precondition 1: If a process calls nextEpoch(S) at the beginning of some cycle c, for some set of servers S then every process that calls nextConsensus during c must have previously called nextEpoch(S) at the beginning of this cycle.
- Precondition 2: Let S be the set of servers of some epoch E, and S´ be the set of servers of epoch E-1. If s is in S-S´ then s enters epoch E only after its Upper

Consensus layer has been initialized with the state that one of the remaining processes p of S has immediately before p enters epoch E.

Given the above preconditions, the same set of properties holds as in the static case, with the exception that Liveness 1 and 2 are slightly modified as follows:

- Liveness 1: If there is a last infinite epoch E, a correct process (of E) execute addProposal(s) during E, and every correct process (of E) p eventually calls nextConsensus again after the previous invocation of nextConsensus by p returns then, eventually, some correct process (of E) q u-decides s.
- Liveness 2: If there is a last infinite epoch E and a process p completes its i-th u-cycle during E then if a correct process (of E) q enters its i-th u-cycle then q completes its i-th u-cycle.

Implementation Overview:

The getState method of this layer captures the internal state of this layer, calls the getState method of the Core Consensus layer, and returns both states encapsulated in a single snapshot object. In an analogous manner, the setState method receives as an argument an object s that encapsulates the states s_c, s_u of a Core and an Upper Consensus layers, respectively, overwrites the Upper Consensus layer's internal state with s_u, and calls the setState method of the underlying Core Consensus layer with argument s_c. The nextEpoch method changes the set of processes that the Upper Consensus layer regards as servers, and calls the nextEpoch method of the Core Consensus layer.

It is not possible that the set of server changes in the middle of the execution of a consensus, because nextEpoch is not executed concurrently with any execution of nextConsensus by the same process. From Preconditions 1 and 2 (which as we will see in Section 5.2.4.1 are guaranteed by the implementation of the Server Management layer), and the inductive hypothesis that the properties of static case hold until the last cycle of epoch E-1 it follows that the same properties still hold in epoch E, since at the beginning of epoch E all servers have exactly the same history. Note that Preconditions 1 and 2 of the Core Consensus layer are satisfied by the corresponding preconditions of the Upper Consensus layer and the fact that the getState method of Core Consensus resets the R-sets of Core Consensus.

5.2.4 Management Layer

5.2.4.1 Server Management layer

The Management layer of some server s in the dynamic case has the following responsibilities in addition to those it has in the static case. It periodically queries the Failure Detector layer about the health level of every server. If the Failure Detector of a server s reports some of the other servers to be *seriously suspected*, i.e., their health level is reported to be equal to minHelthLevel, then the Management layer creates a suspicion report and passes it to the Upper Consensus layer. Moreover, when the SGM module orders an epoch change the Management layer of s is responsible for getting a snapshot of the state of the several layers of s and sending that snapshot to the new members of the

server group. If s is not a member of the server group of the next epoch the Management layer is responsible for transforming s to an agent.

Interfaces:

The Snapshotable interface described in Section 5.1 is added as a call-back interface in order to allow captuwring and setting the internal state of the attached Upper Consensus and Server Application layers.

Properties:

The same properties as in the static case apply in the dynamic case as well, namely Liveness, Integrity, and Linearizability. The only difference is that the Liveness property is guaranteed only for client requests received by the Management layer of a server during the most recent epoch.

- Liveness: If there is a last infinite epoch E, and p is a correct process (of E) such that the Server Application layer returns a reply for every request passed to it by the Server Management layer then the Server Management layer of p eventually sends a response to the Client Management layer of any client for every request it receives from the Client Management layer of that client.

Implementation Overview:

The Management layer of a server s executes three tasks in parallel. The first task manages client requests and is almost the same as the first task of the Server Management

layer in the static case (Section 4.5.1). The only difference is that a server may also receive client requests relayed by agents in addition to requests received directly by clients.

The second task periodically queries the Failure Detector layer about the health level of every server. If during some epoch E_s the Failure Detector of a server s seriously suspects server s′, i.e., the healthLevel of s′ is equal to the minHealthLevel value, then the following actions are taken. The Management layer of s retrieves the health level of all agents from the Failure Detector layer. Then a special request r, called suspicion report, is generated and passed down to the Upper Consensus layer similarly to normal client requests. Request r contains (i) the health level of all servers and agents as returned by the failure detector, (ii) the current epoch of s (E_s), and (iii) an integer *timestamp*, that uniquely identifies every suspicion report of p and is chosen so that more recent requests have greater timestamps.

The third task invokes the nextConsensus method of the Upper Consensus layer. During the initialization of the task nextConsensus is called. Every time a call of nextConsensus returns an ordered list of requests, the client requests in the list are sequentially (in the order they appear in the list) passed to the appropriate module of the Server Application layer. The corresponding replies are treated as in the static case. Then the suspicion report requests are sequentially (again in the order they appear in the list) passed to the SGM module. If the SGM module responds to a suspicion report request with an “Advance

Epoch Order[12]" reply indicating that the new set of server is S, then the Management layer proceeds as follows. If this is not the first "Advance Epoch Order" reply received in the current epoch then the order is discarded. Otherwise, the Management layer, first, gets a snapshot of its own state, and the state of the two Consensus layers bellow and all the Server Application modules above by invoking the getState methods of Upper Consensus and Application layers. These snapshots are sent to all members of S-S´, where S´ is the current set of servers. Then, if the process is still in S, it invokes the nextEpoch method of Upper Consensus layer with argument S, and when the execution of nextEpoch is completed the nextConsensus method of the Upper Consensus layer is called. Processes in S´-S start acting as agents.

All the three properties of this layer follow by the same reasoning as the corresponding properties of the Server Management layer in the static case. The proofs only differ in that the modified versions of Liveness are used. In order to use the properties of Upper Consensus it is required that the two preconditions described in Section 5.2.3 be satisfied. Assuming that Precondition 1 holds until some cycle c, then from the safety properties of Upper Consensus (Validity 1, 2, and Uniform Agreement) it follows that the Upper Consensus layer of every server passes the requests (in particular, the suspicion report requests) to the Management layer in the same order. Then, from the Server Management layer protocol sketched above and the determinism of the GSM module, it follows that Precondition 1 holds for cycle c+1. Precondition 2 follows from the Server and Agent Management layer protocol (the latter is described in then next section.) and the fact that

[12] A more appropriate name is "Advance Epoch Command"

for any two consecutive epochs there exists at least one process p that is a server in both E and E+1 and does not crash during both E and E+1.

5.2.4.2 Agent Management layer

The Management layer of an agent has the same structure as the Management layer of a server, but its responsibilities are restricted to the following tasks. First, it forwards client requests it receives from a client to some process the agent believes is currently a server – if this belief is incorrect then the request is lost and it is up to the client's Management layer to resend the request to different agent or server. Second, when instructed, an agent becomes a server. (Since the set of agents/servers is fixed, if the client is resistant enough, it will eventually succed in sending the request to an actual server.) Regarding the number of times an agent can switch to a server and back we consider two cases. Either a bounded number of switches is allowed per process, or no bound is imposed. We call the two cases as *finite dynamic case* and *infinite dynamic case*, respectively.

Interfaces:

The Management layer of an agent has exactly the same interfaces as the corresponding layer of a server.

Implementation Overview:

When an agent receives a snapshot object from the Management layer of a server then it overwrites its own state, the state of its Upper and Core consensus layer by calling the setState method of Upper Consensus, and the state of the Application layer by calling the setState method of the Application layer. The new states are the ones encapsulated in the received snapshot object. Then it calls the nextEpoch method of Upper Consensus, and, when this call returns, it calls the nextConsensus method of Upper Consensus. This last call marks the switch of the agent to a server. Note that agents and servers typically store any messages they receive and appear to belong to future cycles. This fact still holds in the case where some old servers who have already learned about the new one start sending messages to it (say as part of the consensus) before the agent has received the state snapshots and installed it. In such a case these messages are stored on the agent and processed once the agent has fully become a server. Moreover, only the first snapshot object received since the process became an agent for the last time is used. If a server receives a snapshot object of a server in a future epoch then it overwrites its state accordingly the same way an agent does.

5.2.4.3 Client Management

The Client Management layer does not distinguish between servers and agents; it chooses the process to which it sends its request among the set of all servers and agents alike. In contrast to the static case, the Client Management layer submits each client request repeatedly until it receives a response. In particular, initially a request is sent to the server that responded to the most recent request of this client. If no response is received within a

timeout the same request is sent to the next process from the set of servers/agents, assuming a random cyclic ordering of agents/servers.

Properties:

We consider two approaches. In the finite dynamic case the same properties as in the static case hold. In the infinite dynamic case only the Linearizability property holds and a decent effort is made to satisfy the liveness property.

Implementation Overview:

In the finite dynamic case, Liveness follows from the Liveness property of the Server Management layer, the Liveness property of the Server Application layer, the Liveness property of the Communication layer, the fact that the Management layer of the client repeatedly sends a request to all servers until it receives a reply, and the observation that eventually all correct servers will reach the same final epoch.

Linearizability follows from the same argument as in the static case.

5.2.5 Server Application Layer

The properties of the Server Application layer do not change, and the interfaces are extended with the addition of the Snapshotable interface. However, in the dynamic case we introduce a special application module, namely the *Server Group Membership Application (SGM)*, in addition to the standard application specific modules. The

existence of this application is transparent to the other server application modules, just like every application is invisible to every other.

Interfaces:

The Snapshotable interface described in 5.1 is added to every module of the Server Application layer to allow capturing and setting the internal state of the modules.

Properties:

The Server Application layer in the dynamic case can satisfy the Liveness property only if there exists a last infinite epoch:

- Liveness: If there is a last infinite epoch E, any call to the applyRequest method of the Server Application layer of a correct process (of E) returns a Reply.

SGM Server Application module:

Intuitive Description:

The SGM Server Application module is responsible for deciding when an epoch change should occur. Recall that the purpose of epoch changes is to replace the processes in the current set of servers that are considered to be faulty with others that are believed to be correct. The Management layer of each server occasionally passes to SGM suspicion reports, which reflect the view of processes about the correctness status of the other

processes. Using this information SGM decides what changes, if any, should be made to the set of servers, and passes the appropriate reply back to the Management layer.

Properties:

- SGM changes the group of servers only once for a given epoch.
- Every time SGM orders an epoch change, the set of servers changes by at most t processes.

Implementation Overview:

A suspicion report is a tuple <p, e_p, ts_p, H_p>, where p is the ID of the process that sends this request; e_p is the epoch of p at which the request is issued; ts_p is an integer *timestamp*, that uniquely identifies every suspicion report of p and is chosen so that more recent requests have greater timestamps; and H_p is a list that contains the health level of all processes (among the servers and agents) as reported by the failure detector of p.

The SGM module of a process p maintains a local set $SuspicionReports_p$, where the received suspicion reports are stored. $SuspicionReports_p$ is updated by SGM in such a way so that it only contains suspicion reports of the current epoch, and for each (server or agent) process it contains just the most recent report received by that process.

When an update in $SuspicionReports_p$ takes place, p decides whether to initiate a new epoch as follows. If no server is found to be seriously suspected (i.e., its health level is

equal to minHealthLevel) by a majority of servers, then the epoch does not change. Otherwise, at most t of the servers seriously suspected by some majority of servers are chosen to be replaced. If more than t such servers exist, the ones suspected by the most processes are chosen. Then, SGM computes the *cumulative health level* of every agent p as the sum of the health levels of p in all reports. The servers chosen to be replaced, are substituted in the current set of servers **S** by the agents of highest cumulative health level. A new epoch is initiated by passing down the management layer an "Advance Epoch Order" reply containing the new set of servers. This approach is based on the intuition that a process assumed to be crashed by a large number of other processes is more likely to be faulty than a process believed to be in good health by many processes. The requirement that at least t+1 processes must seriously suspect a server in order for this server to be replaced prevents a minority of servers with bad communication with the rest of the servers to remove from the set of servers healthy processes. On the other hand, the Competeness property of the ◊S failure detector guarantees that a faulty process will eventually be suspected by all correct processes and, thus, will be eventually replaced. The restriction that at most t membership changes may occur in S when an epoch changes is needed to ensure liveness if more than t processes are chosen to be replaced then all the at most t processes that are common to both the old and new set of servers may crash before any new server receives an snapshot of the state required for its initialization, causing the system's operation to block.

Chapter 6

Conclusions – Future Work

Our system was designed and implemented using the Java programming language (SDK version 1.4.2). Each layer of the system was implemented as a distinct Java package, consisting of various classes and interfaces. Each class was thoroughly tested to ensure correctness, using both black box and white box techniques and carefully selected test vectors. In most cases we needed to write additional code to accomplish the testing. This code is preserved, but disabled, in the source files to help in testing of future versions of the software.

Moreover, the whole system was tested by applying a Client Application module which sends requests in the form of (op, value), where op is an arithmetic operation randomly chosen from the set {+, -, ·, ÷}, and value is a random integer number (÷ is integer division). The configuration we used in this application was t=1, n=4, which implies that

every epoch consisted of 3 servers and 1 agent. All processes were networked on a TCP/IP local area network.

Due to time restrictions, we did not implement the snapshotable interface. Instead, agents reconstruct the state of servers by incrementally receiving the order of requests from the servers and passing the requests to their Server Application layers. If an agent receives a message to become server and it has not yet constructed all the needed state, it broadcasts messages asking for the missing requests. This approach affects only the way the system's state is transferred to agents and it does not change the way the sets of agents and servers are maintained by the SGM module, or the way the Management layer of the dynamic case works.

A characteristic of the system is that the whole history of request-reply pairs must be stored to make sure that each client request is executed only once and that every client eventually receives the replies of all the requests it submits. This feature impacts on the performance of the system since the accumulation of requests increases the size of the data structures and consecutively increases the time the program needs to manipulate them. Therefore, it is interesting to find ways to remove old requests from system's memory, while preserving the properties provided to the client.

Throughout this work we assumed that the maximum number of servers that can crash during an epoch is constant. In some systems there might be important phases of execution that need increased resiliency in faults, so it would be interesting to study a

dynamic case in which each epoch can differ from the previous not only on the group of servers but also on the resiliency level t.

The protocol used in the Core Consensus layer is linear in terms of message transmissions, but it needs 6 stages to complete. There exist consensus protocols with quadratic message-complexity that decide in fewer stages. Since our system is designed to use only a portion of the available processes to run the consensus protocol, the use of such protocols by only few servers (for instance only three) could result in the production of fewer total messages. Such a system could prove to be faster, and it would be interesting to implement a Core Consensus layer based on a fast quadratic algorithm and compare the two cases in practice.

Finally, the incorporation of a failure detector that provides quality of service guarantees and the study of the impact that it would have on the properties and behaviour of the system is an interesting problem for future research.

Bibliography

[1] T. Chandra and S. Toueg., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, March 1996.

[2] U. Fritzke, P. Ingels, A. Mostefaoui and M. Raynal, Fault-Tolerant Total Order Multicast to Asynchronous Groups. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, Purdue University, pp. 228-235, October 1998.

[3] R. Guerraoui, Revising the Relationship between Non-Blocking Atomic Commitment and Consensus. *Proc. 9th Int. Workshop on Distributed Algorithms (WDAG95)*, Springer-Verlag LNCS 972 (J.M. Helary and M. Raynal Eds), pp. 87-100, September 1995.

[4] M.J. Fischer, N. Lynch, M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, April 1985.

[5] T. D. Chandra, V. Hadzilacos and S. Toueg, The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, July 1996.

[6] A. Schiper, Early Consensus in an Asynchronous System with Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997

[7] M. Hufin, M. Raynal. A Simple and Fast Asynchronous Consensus Protocol Based on a weak failure detector. *Distributed Computing*, 12(4), 1999

[8] A. Mostéfaoui , M. Raynal, Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based Approach, *Proc. of the 13th Int. Symposium on Distributed Computing*, pp. 49-63, September 1999.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7):558-565, July 1978

[10] D. Ivan and S. Toueg. Uniform Atomic Broadcast Algorithms. *Unpublished manuscript.* 2002.

[11] F. Cristian, H. Aghili, H.R. Strong, D. Dolver. Atomic Broadcast: From simple message diffusion to Byzantine Agreement. *Proc of 15th Int Symposium on Fault-Tolerant Computing, pp 200-206, June 1985.*

[12] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*. Volume 448 of *LNCS*, pp. 201-208. Springer-Verlag, 1987.

[13] A. Schiper and S. Toueg. From Set Membership to Group Membership: A Separation of Concerns. *EPFL Technical Report 200371*. November 2003.

[14] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393-481. Springer-Verlag, 1978. Also appears as Technical Report RJ2188, IBM Research Laboratory.

[15] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Computer Science Laboratory, Xerox, Palo Alto Research Center ,Palo Alto, CA. 1976.

[16] D. Skeen. Crash Recovery in a Distributed Database System. PhD thesis, U.C. Berkeley, May 1982.