A MODULAR MIDDLEWARE FOR RELIABLE DISTRIBUTED PROGRAMMING *

Mikel Larrea, Alberto Lafuente, Cristian Martín
The University of the Basque Country
20018 San Sebastián, Spain

ABSTRACT
This paper presents a modular middleware for reliable distributed programming, formed by the following modules: failure detection, reliable point-to-point communication, reliable broadcast, Consensus, and total order broadcast. The lower layer of the middleware is built on top of an unreliable message-passing service. The upper layer, namely total order broadcast, provides the semantics required for the programming of reliable distributed applications based on active replication. We have implemented and evaluated a prototype of the middleware in the Java programming language.

KEYWORDS
Distributed systems, fault-tolerance, total order broadcast.

1. INTRODUCTION

The advent of low-cost, off-the-shelf computing hardware, combined with the development of networking infrastructures are making distributed computing more and more important nowadays. An appealing feature of distributed systems, due to its inherent redundancy, is its potential resiliency to partial failures. In a distributed system, processes and/or data can be replicated on several computers, such that the failure of one computer does not affect the functioning of the system. However, replication has an important drawback, since extra complexity has to be introduced for consistency management. Depending on the correctness criteria required by the application, several consistency levels can be defined. The most common consistency criteria are sequential consistency and linearizability, being the last one stricter. Linearizability is the criteria required by, for example, fault-tolerant banking systems.

Two different techniques can be used to provide linearizability in a replicated system: passive replication (also called primary-backup) and active replication [6]. Both approaches are usually implemented based on the services provided by an underlying group communication infrastructure. Passive replication relies on the control that a particular replica (the primary) makes on the system. This centralized, synchronous approach, demands from the group communication system some form of reliable broadcast between the primary and the backups, but failures must be explicitly managed (e.g., electing a new primary). Even more, in a particular failure scenario clients should cope with request re-play, which jeopardizes failure transparency. On the other hand, active replication demands a much heavier group communication support, namely total order broadcast, but full failure transparency is provided to clients.

A usual approach to support the programming of reliable distributed applications consists on decoupling the logic of the application itself from an underlying infrastructure, also known as middleware, that provides the services required for fault-tolerance. In this paper, we present such a middleware for reliable distributed programming, composed by five modules: (1) failure detection, (2) reliable point-to-point communication, (3) reliable broadcast, (4) Consensus, and (5) total order broadcast. The lower layer of the middleware, formed by failure detection and reliable point-to-point communication, is built on top of an unreliable service.

message-passing service, typically provided by the underlying operating system, e.g., UDP/IP. The upper layer of the middleware, namely total order broadcast, provides the semantics required for the programming of reliable distributed applications based on active replication. We have implemented a prototype of the middleware in the Java programming language. As the modules (Java packages) can be used separately, our approach allows composing ad-hoc middleware architectures to fit application requirements (e.g., to support passive replication based on reliable broadcast). We have evaluated the performance of the Consensus and total order broadcast protocols in a local area network and in Internet.

Several middleware, providing reliable group communication primitives, have been proposed, e.g., Ensemble, JGroups and Spread. Contrary to our approach, these middleware consider a virtually synchronous system model, which means that they assume reliable failure detection. Hence, the protocols used to implement total order broadcast are not based on Consensus but on a sequencer process.

The rest of the paper is organized as follows. Section 2 presents the system model and gives the specification of the problems corresponding to the modules of the middleware. Section 3 presents the architecture of the middleware and the protocols that have been implemented as the solutions to the problems. Section 4 presents the performance analysis results. Finally, Section 5 concludes the paper.

2. SYSTEM MODEL AND PROBLEM SPECIFICATIONS

2.1 System model

We consider a message-passing distributed system composed of a finite set of \( n \) processes, \( \Pi = \{p_1, p_2, \ldots, p_n\} \). Every pair of processes is connected by an unreliable point-to-point communication channel. Processes can fail by crashing, i.e., by prematurely halting. Crashes are permanent, i.e., crashed processes do not recover.

Except for the failure detection module, we assume that the system is asynchronous, i.e., there are no upper bounds on neither the transmission delay of messages nor the relative speeds of processes. However, it has been shown by Fischer, Lynch and Paterson that the Consensus problem \([9]\) cannot be solved deterministically in an asynchronous system with reliable channels that is subject to even a single crash failure \([4]\). Intuitively, this is because, in such a system, it is impossible to distinguish a very slow process from one that has actually crashed. This result also applies to the total order broadcast problem. Actually, the model of unreliable failure detectors \([2]\) has been proposed to circumvent the FLP impossibility result.

Thus, in order to implement the failure detection module, we assume that the system is partially synchronous \([2, 3]\). In particular, we consider the following model of partial synchrony \([2]\): in every execution, there are bounds on relative process speeds and on message transmission times, but these bounds are not known and they hold only after some unknown (but finite) time GST (for Global Stabilization Time).

2.2 Failure detection

The concept of unreliable failure detector was introduced by Chandra and Toueg in \([2]\) as a mechanism that provides information about process failures. They characterized a failure detector in terms of two properties: completeness, which characterizes the failure detector capability of suspecting incorrect processes, and accuracy, which characterizes the failure detector capability of not suspecting correct processes.

In this paper, we focus on the Eventually Perfect failure detector class, denoted \( \Diamond P \), which satisfies (i) Strong Completeness: eventually every process that crashes is permanently suspected by every correct process, and (ii) Eventual Strong Accuracy: there is a time after which correct processes are not suspected by any correct process. Chandra and Toueg showed in \([2]\) that \( \Diamond P \) can be used to solve the Consensus problem in an asynchronous system with a majority of correct processes.\(^2\)

\(^1\) We assume fair lossy channels, i.e., channels that may lose a subset of the messages that are sent, but not all of them.

\(^2\) Actually, a weaker class of failure detectors, named Eventually Strong and denoted \( \Diamond S \), is sufficient to solve Consensus. The class \( \Diamond S \) satisfies Strong Completeness and Eventual Weak Accuracy, defined as follows: there is a time after which some correct process is not suspected by any correct process.
2.3 Reliable point-to-point communication

Reliable point-to-point communication for asynchronous systems is a basic communication service of our middleware. Informally, it guarantees that (1) all messages sent between correct processes are delivered, and (2) no spurious messages are ever delivered. Formally, reliable point-to-point communication is defined in terms of two primitives, R-send($m$) and R-receive($m$). We assume that every message $m$ includes a field denoted sender($m$) that contains the identity of the sender, and a field with a sequence number; these two fields make every message unique. Reliable point-to-point communication satisfies the following properties: (i) Reliability: if a correct process $p$ R-sends a message $m$ to a correct process $q$, then $q$ eventually R-receives $m$, and (ii) Uniform integrity: for any message $m$ R-sent to a process $q$, $q$ R-receives $m$ at most once, and only if $m$ was previously R-sent by sender($m$).

Note that the reliability property is restricted to pairs of correct processes. From a theoretical point of view, stronger forms of reliability have been defined [1], that guarantee the R-reception of a message when the destination is correct but the sender may fail (obviously, assuming that the sender has R-sent the message before failing). Usually, reliability is achieved using retransmission techniques (the message is retransmitted by the sender until the receiver, which we assume is correct, acknowledges it). The potential crash of the sender makes difficult the implementation of such a channel model. In practice, distributed protocols are usually designed based on the weaker correct-restricted reliable channel model, as defined above.

2.4 Reliable broadcast

Reliable broadcast for asynchronous systems is another communication primitive of our middleware. Informally, reliable broadcast guarantees that (1) all correct processes deliver the same set of messages, (2) all messages broadcast by correct processes are delivered, and (3) no spurious messages are ever delivered. Formally, reliable broadcast is defined in terms of two primitives, R-broadcast($m$) and R-deliver($m$). Reliable broadcast satisfies the following properties [7]: (i) Validity: if a correct process R-broadcasts a message $m$, then it eventually R-delivers $m$, (ii) Agreement: if a correct process R-delivers a message $m$, then all correct processes eventually R-deliver $m$, and (iii) Uniform integrity: for any message $m$, every process R-delivers $m$ at most once, and only if $m$ was previously R-broadcast by sender($m$).

2.5 Consensus

In the Consensus problem, each process initially proposes a value, and all correct processes must reach an irrevocable decision on some common value that is equal to one of the proposed values. Formally, the Consensus problem is defined in terms of two primitives, propose and decide. The Consensus problem is specified as follows: (i) Termination: every correct process eventually decides some value, (ii) Uniform integrity: every process decides at most once, (iii) Agreement: no two correct processes decide differently, and (iv) Uniform validity: if a process decides $v$, then $v$ was proposed by some process. Termination defines the liveness property associated with the Consensus problem, while Uniform integrity, Agreement and Validity define its safety properties.

The Agreement property allows faulty processes to decide differently from correct processes. This fact can be sometimes undesirable as it does not prevent an incorrect process to propagate a different decision throughout the system before crashing. In the Uniform Consensus problem, agreement is defined by the following property, which enforces the same decision on any process that decides: no two processes (correct or faulty) decide differently. It has been shown in [5] that any protocol that solves Consensus using a failure detector of class $\Diamond S$, also solves Uniform Consensus.

2.6 Total order broadcast

Total order broadcast is a fundamental problem in fault tolerant distributed computing. Informally, total order broadcast requires that all correct processes deliver the same messages in the same order. Formally, total order broadcast is a reliable broadcast that satisfies (iv) Total order: if two correct processes $p$ and $q$ deliver two messages $m$ and $m'$, then $p$ delivers $m$ before $m'$ if and only if $q$ delivers $m$ before $m'$. 
The total order and agreement properties of total order broadcast ensure that all correct processes deliver the same sequence of messages. Note that reliable broadcast ensures that all correct processes deliver the same set of messages. This apparently small difference (i.e., sequence versus set) has a tremendous impact from a solvability point of view of both problems: while reliable broadcast can be solved deterministically in an asynchronous system subject to crash failures, total order broadcast cannot.

Chandra and Toueg showed in [2] that Consensus and total order broadcast are equivalent problems in asynchronous systems with crash failures. This means that total order broadcast can be solved whenever Consensus can be solved.

3. THE ARCHITECTURE AND THE PROTOCOLS

Figure 1 presents the architecture of the middleware. The failure detection and reliable point-to-point communication modules are based on an unreliable message-passing service, e.g., UDP/IP, typically provided by the underlying operating system. The reliable broadcast module uses only reliable point-to-point communication. The Consensus module uses failure detection, reliable point-to-point communication and reliable broadcast. Finally, the total order broadcast module uses reliable broadcast and Consensus.

Fault tolerant applications (by active replication)

Total order broadcast

Consensus

Reliable broadcast

Failure detection

Reliable point-to-point communication

Unreliable point-to-point communication (e.g., UDP/IP)

Figure 1. Architecture of the middleware

Concerning the protocols, for failure detection we have implemented the heartbeat based protocol proposed by Chandra and Toueg [2] implementing $◊P$ in models of partial synchrony. For reliable point-to-point communication, we have implemented a protocol based on message retransmissions and the use of ACKs to acknowledge the correct reception of messages. For reliable broadcast, we have implemented the diffusion based protocol proposed by Chandra and Toueg [2]. For Consensus, we have implemented the $◊S$-Consensus protocol proposed by Chandra and Toueg [2]. Since the class $◊P$ is stronger than $◊S$, this protocol remains correct on top of any failure detector of class $◊P$. The protocol uses the rotating coordinator paradigm, and it proceeds in asynchronous “rounds”. It is guaranteed that eventually a correct coordinator will succeed in taking a decision and broadcasting it to the rest of correct processes. This protocol relies on the necessary assumption that there is a majority of correct processes in the system. Finally, for total order broadcast we have implemented the protocol proposed by Chandra and Toueg [2]. This protocol uses repeated (possibly concurrent, but completely independent) executions of Consensus to decide on the next batch of messages to be totally ordered delivered.

More details about the protocols can be found in [8].
4. PERFORMANCE ANALYSIS

We have implemented the middleware in the Java programming language and environment. Each module has been implemented as a Java package. We present here some preliminary results of the performance analysis of the Consensus and total order broadcast protocols. In all the tests, we have considered the favorable case in which there are neither faults nor incorrect suspicions reported by the failure detector, which is the most common scenario in practice. To do so, we have tuned the parameters of the failure detection module in a way that the probability of making a mistake is negligible. This way Consensus is always solved in the first round.

We have tested the protocols with up to 11 nodes in a non-isolated 10 Mbps Ethernet LAN, and with 3 nodes in Internet. In the case of Consensus, we have executed 10000 consecutive Consensus, measuring for each one of them the elapsed time between propose and decide. In the case of total order broadcast, each process has broadcast 5000 messages, which gives a total of $5000n$ messages to deliver totally ordered, being $n$ the number of processes in the group. We have measured the number of Consensus required to deliver all these messages, the number of messages delivered within each Consensus, and the time required by each Consensus. These measures allow us to derive an average time of message delivery. Figures 2, 3 and 4 show the results for the local area network scenario. Figure 4 shows that almost half of the Consensus deliver between 1 and 10 messages.

<table>
<thead>
<tr>
<th>#nodes</th>
<th>t_min_Cons</th>
<th>t_max_Cons</th>
<th>t_avg_Cons</th>
<th>#Cons/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8 ms</td>
<td>65 ms</td>
<td>16,2 ms</td>
<td>61,7</td>
</tr>
<tr>
<td>5</td>
<td>11 ms</td>
<td>69 ms</td>
<td>21,8 ms</td>
<td>45,9</td>
</tr>
<tr>
<td>7</td>
<td>14 ms</td>
<td>99 ms</td>
<td>30,1 ms</td>
<td>33,2</td>
</tr>
<tr>
<td>9</td>
<td>19 ms</td>
<td>98 ms</td>
<td>42,0 ms</td>
<td>23,8</td>
</tr>
<tr>
<td>11</td>
<td>25 ms</td>
<td>132 ms</td>
<td>48,2 ms</td>
<td>20,7</td>
</tr>
</tbody>
</table>

Figure 2. Performance of Consensus in a local area network

<table>
<thead>
<tr>
<th>#nodes</th>
<th>#msg</th>
<th>#Cons</th>
<th>msg/Cons</th>
<th>t_avg_Cons</th>
<th>t_avg/msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15000</td>
<td>3313</td>
<td>4,5</td>
<td>34,1 ms</td>
<td>7,6 ms</td>
</tr>
<tr>
<td>5</td>
<td>25000</td>
<td>2229</td>
<td>11,2</td>
<td>50,9 ms</td>
<td>4,5 ms</td>
</tr>
<tr>
<td>7</td>
<td>35000</td>
<td>2457</td>
<td>14,2</td>
<td>62,4 ms</td>
<td>4,3 ms</td>
</tr>
<tr>
<td>9</td>
<td>45000</td>
<td>2441</td>
<td>18,4</td>
<td>73,2 ms</td>
<td>3,9 ms</td>
</tr>
<tr>
<td>11</td>
<td>55000</td>
<td>3754</td>
<td>14,6</td>
<td>72,1 ms</td>
<td>4,9 ms</td>
</tr>
</tbody>
</table>

Figure 3. Performance of total order broadcast in a local area network

Figure 4. Number of messages delivered by Consensus (11 nodes)
For the tests in Internet, we have used the following three nodes in Spain: (1) UPV/EHU (Universidad del País Vasco), located in San Sebastián, (2) URJC (Universidad Rey Juan Carlos), located in Madrid, and (3) UJI (Universidad Jaume I), located in Castellón. Using the ping utility, we have measured the communication delay between the nodes. The results are the following: 16.5 ms between UPV/EHU and URJC, 9.0 ms between URJC and UJI, and 23.5 ms between UPV/EHU and UJI.

Figures 5 and 6 show the results in Internet. For Consensus, the average time is 7 times bigger than in a LAN. For total order broadcast, the average number of messages delivered by each Consensus increases from 4.5 in a LAN to 19.2 in Internet, a consequence of the increase of the time required to solve Consensus.

<table>
<thead>
<tr>
<th>#nodes</th>
<th>t_min_Cons</th>
<th>t_max_Cons</th>
<th>t_avg_Cons</th>
<th>#Cons/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>60 ms</td>
<td>240 ms</td>
<td>113 ms</td>
<td>8.85</td>
</tr>
</tbody>
</table>

Figure 5. Performance of Consensus in Internet

<table>
<thead>
<tr>
<th>#nodes</th>
<th>#msg</th>
<th>#Cons</th>
<th>#msg/Cons</th>
<th>t_avg_Cons</th>
<th>t_avg/msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15000</td>
<td>780</td>
<td>19,2</td>
<td>193.6 ms</td>
<td>10.1 ms</td>
</tr>
</tbody>
</table>

Figure 6. Performance of total order broadcast in Internet

5. CONCLUSION

In this work, we have presented a modular, composable middleware architecture to support the programming of fault-tolerant applications by active replication. The middleware is composed of the following modules: failure detection, reliable point-to-point communication, reliable broadcast, Consensus, and total order broadcast. We have implemented a prototype and evaluated its performance by series of executions of Consensus and total order broadcast. Although these are inherently communication-expensive protocols, the results obtained show that reliable distributed applications based in our architecture seem to be viable in some scenarios, e.g., local area networks.

A number of improvements may be foreseen for the future, namely adding new modules to the middleware (e.g., atomic commit, leader election …), and offering alternative protocols to current modules (e.g., Ω and ◊ failure detectors, alternative Consensus and total order broadcast protocols …). Besides, more experimentation should be carried out to evaluate performance, especially in wide area network scenarios.

REFERENCES