IOPscience iopscience.iop.org

Home Search Collections Journals About Contact us My IOPscience

Supporting customized failure models for distributed software

This article has been downloaded from IOPscience. Please scroll down to see the full text article.

1999 Distrib. Syst. Engng. 6 103

(http://iopscience.iop.org/0967-1846/6/3/302)

View the table of contents for this issue, or go to the journal homepage for more

Download details:
IP Address: 38.107.179.212
The article was downloaded on 20/02/2012 at 20:57

Please note that terms and conditions apply.

Distrib. Syst. Engng 6 (1999) 103–111. Printed in the UK PII: S0967-1846(99)07015-1

Supporting customized failure models for distributed software

Matti A Hiltunen, Vijaykumar Immanuel† and Richard D Schlichting

Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA

Received 16 August 1999

Abstract. The cost of employing software fault tolerance techniques in distributed systems is strongly related to the type of failures to be tolerated. For example, in terms of the amount of redundancy required and execution time, tolerating a processor crash is much cheaper than tolerating arbitrary (or Byzantine) failures. This paper describes an approach to constructing configurable services for distributed systems that allows easy customization of the type of failures to tolerate. Using this approach, it is possible to configure custom services across a spectrum of possibilities, from a very efficient but unreliable server group that does not tolerate any failures, to a less efficient but reliable group that tolerates crash, omission, timing, or arbitrary failures. The approach is based on building configurable services as collections of software modules called micro-protocols. Each micro-protocol implements a different semantic property or property variant, and interacts with other micro-protocols using an event-driven model provided by a runtime system. In addition to facilitating the choice of failure model, the approach allows service properties such as message ordering and delivery atomicity to be customized for each application.

1. Introduction

Distributed architectures are increasingly used to construct systems that must continue to operate despite failures such as processor crashes. Unfortunately, providing *fault tolerance* of this type can be expensive. For example, one strategy is to replicate the application on multiple independent machines and then operate them in logical synchrony using the state machine approach [1]. This requires not only redundant hardware, but also sophisticated underlying software such as group atomic multicast [2, 3] to keep the states of the replicas synchronized.

One factor determining the cost of providing fault tolerance in distributed systems is the type and number of failures to be tolerated. Typically, the type of faults to be tolerated is expressed in terms of *failure models* that range from relatively benign crash or omission failures to arbitrary (or Byzantine [4]) failures. In terms of the amount of redundancy required and execution time, tolerating a processor crash is much cheaper than tolerating Byzantine failures. Furthermore, the cost is determined not only by the failure model, but in many cases also by the number or frequency of the failures expected to occur. For example, a simple replication scheme intended to tolerate n crash failures requires $n + 1$ replicas.

Any realistic system can, in principle, exhibit failures in any of the failure models but, typically, failures in the more benign classes are more frequent than severe failures. Thus, the choice of failure model for a particular task should be based on the frequency of different types of failures in the given execution environment, as well as the criticality of the task. The tradeoff, of course, is that making stronger assumptions about failures improves the performance of the system, but lessens the degree of fault coverage provided by the system and thus the reliability of the task [5].

This paper describes an approach to constructing configurable services for distributed systems that allows easy customization of the type of failures to be tolerated. For example, using our approach, it is possible to configure custom services such as communication services across a spectrum of possibilities, from a very efficient but unreliable service that does not tolerate any failures, to a less efficient but reliable service that tolerates crash, omission, timing, or arbitrary failures. The approach is based on building configurable services using software modules called *micro-protocols*. Each micro-protocol implements a different semantic property or property variant, and interacts with other micro-protocols using an event-driven model provided by a runtime system. The net result is an enhanced ability to explicitly manage the tradeoff between the level of reliability and cost. As an example, we apply the approach to a group remote procedure call (GRPC) service that can be used by a client to transmit a request to a group of replicated servers. In addition to facilitating the choice of failure model, our approach allows GRPC service properties such as message ordering and delivery atomicity to be customized for each application.

† Present address: Compaq Computer Corporation, 19333 Vallco Pkwy, Cupertino, CA 95014, USA.

 © 1999 The British Computer Society, The Institution of Electrical Engineers & IOP Publishing Ltd

2. Customizing fault tolerance

2.1. Assumptions

We consider a distributed system model in which multiple hosts with no shared memory are connected by a communication network. We assume the underlying communication service is similar to one provided by the UDP protocol in a typical Ethernet or Internet environment. That is, we assume that communication is unreliable and unordered, and that the end-to-end transmission latency is typically predictable, but that some messages may be delayed beyond this point due to collisions or network congestion. Hosts can only interact by sending and receiving messages through this communication network. Hosts may fail in arbitrary ways consistent with the assumed failure model, but we assume that the communication network does not experience permanent failures. Among other things, this implies that partitions are not considered.

Given this model, a faulty host can only affect other hosts by not sending a message when it should, by sending a message when it should not, or by sending an incorrect message. A faulty host may exhibit the incorrect behaviour only to a subset of the other hosts.

These kinds of incorrect behaviours can easily be mapped to traditional failure model definitions, including crash, omission, timing, value and Byzantine:

- *Crash.* A host may permanently halt and hence, fail to send messages. If a host is in the process of sending a message when it crashes, some of the intended receivers may not receive the message.
- *Omission.* A host may repeatedly and irregularly fail to send a message to all or some of the intended receivers, or fail to receive messages.
- *Timing.* A host may send a message earlier or later than expected. The network delaying a message longer than expected may result in a host appearing to have a late timing failure.
- *Value.* A host may send a message with incorrect contents. We assume, however, that if a value faulty host sends a multicast message, all receivers will receive the same message contents, i.e. we assume value failures are symmetric [6].
- *Byzantine.* A host may do anything. This means that in addition to failures of the value and timing type, a faulty host may deliberately attempt to confuse other hosts by sending different versions of a message to different receivers or by impersonating another host.

2.2. Implementation approach

Our approach to providing configurable fault tolerance is based on implementing distributed services using Cactus [7, 8]. In Cactus, services are implemented as *composite protocols* constructed from fine-grain software modules called micro-protocols. Each micro-protocol implements a well-defined property or functional component of a service. A customized service variant is configured by combining the micro-protocols that implement the desired properties and functionality. The goal of this paper is to demonstrate that the Cactus model can easily be used to construct micro-protocol suites from which various composite protocols tolerating different classes of failures can be configured.

The Cactus model supports configurability by providing mechanisms—events and shared variables—that maximize the independence between micro-protocols. In this model, a micro-protocol is structured as a collection of *event handlers* that are bound to events signalling state changes of interest, e.g. 'message arrival from the network'. When an event occurs, all event handlers bound to that event are executed. Due to the event mechanisms, micro-protocols can cooperate without directly invoking methods or operations on one another, which makes them more independent. Execution of handlers is atomic with respect to concurrency, i.e. each handler is executed to completion before execution of the next handler is started. Such atomicity eliminates most concurrency control problems associated with access to shared variables.

Event handler binding, event detection, and invocation are implemented by the Cactus runtime system, which is linked with selected micro-protocols to form a composite protocol implementing the custom service. The two primary event-handling operations are `bind()`, which specifies a handler to be executed when a specified event occurs, and `raise()`, which raises a specified event with either blocking or non-blocking invocation semantics. Certain events are also raised implicitly by the Cactus runtime system, such as the message arrival event above. Other operations are available for creating and deleting events, stopping event execution, cancelling a delayed event, and unbinding event handlers from events.

Cactus has been implemented on the OSF/RI MK 7.3 Mach operating system and CORDS [9], a variant of the *x*-kernel system for building network software [10]. On this platform, each composite protocol exports the same external uniform protocol interface (UPI) as CORDS protocols, which allows them to be combined transparently with CORDS protocols such as IP or UDP to form the protocol stack. Communication between CORDS protocols is based on a push/pop paradigm, where a higher-level protocol pushes a message to a lower-level protocol and a lower-level protocol pops a message to a higher-level protocol.

2.3. Providing fault tolerance

The focus of this paper is on constructing configurable distributed services that can exploit existing algorithms for tolerating different classes of failures. The ideal would be to encapsulate such algorithms as 'fault tolerance modules' that could be combined with any distributed service S to automatically create a version of S that could tolerate a selected failure type. While feasible in certain cases [11, 12], our goal is instead to provide fault tolerance modules—implemented as Cactus micro-protocols—that are specific to a given service such as atomic multicast [2, 3], group RPC (GRPC) [13, 14], group membership [15, 16], or distributed transactions [17].

The semantic differences between distributed services are the main motivation for constructing service-specific micro-protocols. For example, in GRPC, a non-faulty server

is expected to send a reply message, but no such requirement exists for asynchronous primitives such as atomic multicast. This means that the definition of correct behaviour—and thus the definition of faulty behaviour—is service specific. As the result, a micro-protocol whose role is to add fault tolerance to a given service must be constructed with the semantics of that service in mind.

Semantic variations between versions of the same service also affect what fault tolerance modules must do. For example, if a multicast service is designed to provide atomic delivery of messages to all members of a group, a fault tolerance module must handle the case when a message only reaches a subset of intended destinations because the sending host fails while transmitting the message. However, if the multicast does not provide atomic delivery, a fault tolerance module does not need to address such an incomplete transmission. Finally, depending on the specifics of the service, it may be possible to reduce the cost of fault tolerance by piggybacking fault tolerance information on application messages.

In our approach, a distributed service is implemented as a composite protocol configured from both *service micro-protocols* and *fault tolerance micro-protocols*. Service micro-protocols provide the basic functionality of the service and can be written independently of the failure model. Fault tolerance micro-protocols add tolerance to failures and are specific to a given failure model. Fault tolerance micro-protocols are implemented as collections of event handlers bound to particular events used by the service micro-protocols. These events would typically include those indicating the arrival of a message from lower-level protocols, as well as the imminent departure of a message from the composite protocol to the next lower level. The `bind()` operation has an ordering argument that is used to ensure that handlers in the fault tolerance micro-protocols are executed before or after handlers in the service micro-protocols when necessary. This allows, for example, messages from hosts declared faulty to be filtered out. Thus, the event mechanism allows execution of fault tolerance micro-protocols to be transparently interleaved with service micro-protocols as required.

2.4. Fault tolerance micro-protocols

In general, fault tolerance micro-protocols must prevent a faulty host from interfering with the operation of non-faulty hosts. The different failure models dictate what type of interference is to be tolerated. For the crash failure model, a faulty host fails to send a message when it should, which may cause a non-faulty host to wait forever for a message. Thus, a fault tolerance module designed for this type of failure must ensure that any such wait is eventually terminated. For example, in a group-oriented service such as atomic multicast, this can be done by removing the faulty host from the membership. A crash failure can be suspected if no messages are received from a host within a given time period. No agreement between hosts is necessary since crash failures are eventually detected by all other hosts.

Omission failures can be divided into send and receive omission failures. Send omission failures are different from crash failures, since a faulty host may continue to send messages after it becomes faulty. Detecting that messages are missing from a sequence is naturally service specific, but for many services, it can be based on the use of consecutive message sequence numbers. Agreement between hosts is required to distribute local omission detections, since a faulty host may fail to send a message only to some hosts, and thus, all hosts may not locally detect the failure.

Omission to receive failures are more difficult to handle, since a faulty host may mistake the symptoms for send omission failures of other hosts. Thus, distribution of the detection information is not enough and a majority vote is required. In this case, a host can locally only *suspect* the failure of another host until the agreement is completed. A host may suspect its own receive omission failure if it does not receive messages from any other host or if other hosts repeatedly disagree with its local detection.

Timing failures can be similarly divided into early and late timing failures. The concept of a message being early or late is even more service specific than omissions; that is, no standard mechanism such as sequence numbers can be used to detect timing failures. Late timing failures can often be detected by setting a local timer event to fire at the time when a message should arrive. If the message arrives by this time, the timer event is cancelled; otherwise, the event handler declares the expected sender of the message to be faulty. Agreement on late timing failures is similar to agreement on send omission failures, i.e. it is sufficient to distribute the local detection information. However, a host with an early timing failure may suspect the failure of other hosts, so an agreement phase similar to that for receive omission failures is required.

Note that a longer than expected transmission time of the underlying communication service may cause failure detection mechanisms to falsely detect host failures of any of the above types. The probability of such false detections can be reduced by allowing a longer transmission time before a failure is suspected, by using retransmissions, and by using agreement algorithms. Nevertheless, a false detection may still occur and must be dealt with by forcing the given host to simulate failure and recovery to rejoin the group. Such false detections can be minimized by calibrating the timeout values and number of retransmissions for the network environment used. Note that, although maintaining such group membership in an asynchronous system is theoretically impossible [18], practical systems have demonstrated that such heuristics result in operational systems.

All of the above fault tolerance micro-protocols deal with failures in the time domain, meaning that only one is configured into any given service. However, failures in the value domain are orthogonal, so value failure micro-protocols could be used together with any of the above. Detecting value failures is completely service specific. For example, it may be based on inspecting the format of a message or the range of the values if there is some known range of reasonable values. Or, if the service involves replication such as GRPC, the responses from different hosts can be compared to detect value failures. These types of checks can be implemented easily by a micro-protocol that intercepts and checks messages before they get to the service micro-protocols.

There is no clear-cut boundary between value and arbitrary failures, so typically a value failure micro-protocol would handle a fixed set of value failures, with more extensive failures handled by a micro-protocol dealing with arbitrary failures if desired. An additional possibility raised with arbitrary failures is that a faulty host may try to impersonate another host. Thus, the arbitrary failure micro-protocols add message authentication, i.e. all messages are signed using RSA encryption and the signature is checked at the receiver before the message is forwarded to the service micro-protocols. Byzantine agreement rather than simple voting algorithms is required to agree on host failures. Message atomicity guarantees must also be implemented using Byzantine algorithms.

3. Customizable GRPC service

As an example of how to apply the above principles to a specific distributed service, we present a customizable GRPC service that clients use to access a group of servers. We assume servers maintain data that can be queried and updated, with the consistency requirements for the data depending on the application. Host recovery is not considered.

The basic assumptions about the execution environment are the same as in section 2. We assume a server group of N servers and an arbitrary number of clients, where all servers are assumed to agree on the server group membership, but the clients are not required to have complete server group information.

3.1. GRPC properties

The GRPC service may be customized to guarantee the following properties.

- *Synchronous.* The client is blocked until a response is received or the call is terminated as unsuccessful.
- *Asynchronous.* The client is not blocked and the result of the call is returned to the client application using an upcall to a designated client procedure.
- *Unique.* Each call is executed at each server no more than one time.
- *FIFO.* The calls sent by a given client are processed in the same order by all servers.
- *Atomicity.* All correct servers will execute the same set of calls. This property is guaranteed even if a client does not send a request to all servers in the group.
- *Total order.* All correct servers execute the same calls in the same order.

Total order guarantees that all the correct servers have a consistent state provided that their initial states are consistent.

No guarantees are made concerning faulty clients or servers, except that if a correct server processes a call from a faulty client, all correct servers will process this call if atomicity is guaranteed and all correct servers will process it in a consistent order if total order is guaranteed.

3.2. GRPC failure models

The definitions of faulty behaviour for each failure model in section 2 can be refined since hosts now have specific roles as servers or clients, and the communication patterns are more restricted. For example, a server is omission faulty if it does not provide a reply to a client or if it fails to send any of the periodic messages exchanged between servers. The definition of a value faulty host depends on the semantics of the GRPC. For example, if the state of the servers is supposed to be identical, a server is value faulty if its reply to a call differs from the other replies.

The goal of the GRPC service is to guarantee that correct clients receive a correct result to their calls given, at most, M faulty servers.

3.3. Algorithms

This section outlines the algorithms employed to implement the different options and properties of GRPC.

3.3.1. Communication algorithms. Multicast is implemented as multiple point-to-point UDP transmissions. A host establishes and maintains the state of logical *links* with the hosts with which it wishes to communicate. For example, a client creates a link to each server by using the server's known IP address and UDP port, and a server creates a link with a client by recording the client's IP address and UDP port when it receives the first request. Links between servers are established for ensuring atomicity, total order and similar properties. Since the underlying communication service provides unreliable UDP semantics, the GRPC service includes a reliability component that uses a modified version of the sliding window protocol on each link to provide reliable but unordered delivery of messages. Hosts timeout and retransmit messages a fixed number of times; when this value is exceeded, the host raises a host failure event, notifying other micro-protocols of the suspected failure. Each link is bi-directional and acknowledgements are piggybacked on data messages whenever possible.

3.3.2. Fault tolerance algorithms. The following algorithms are used to handle failures within the different failure classes. Both servers and clients maintain server group membership information. As per our assumptions, the servers must agree on the membership, while each client only maintains its own approximation. Each server is given the initial group membership at startup, and uses failure detection and agreement algorithms to agree on a new membership when failures occur. A client detects that a server is faulty when it fails to receive a response, which may be either a reply or an acknowledgement. The server is declared faulty locally in this case and subsequent requests will not be sent to that server.

Crash, omission, and timing failures are detected by observing the communication from other hosts. In particular, a host is considered failed if it does not acknowledge a message after a fixed number of retransmissions or if a server fails to send a response or forward a request within a certain time period. Note that the same detection mechanism

works for omission and timing failures because it is message specific: i.e., if there is any message that is not acknowledged on time or sent on time, the responsible host is considered faulty.

Although local detection of these failures is similar, the agreement phase varies depending on the failure model. Crash failures do not require any agreement phase, since each host will eventually detect the failure independently. Send omission and late timing failures require the local detection information to be shared between servers. This is done by piggybacking the information on regular messages, which ensures that all non-faulty servers eventually agree on the failure.

Receive omissions and early timing failures require agreement since a faulty host may suspect correct hosts due to a missing message or a fast local clock. Thus, when a failure is detected, the host is declared suspect. Information about the suspected failure is included in the messages exchanged by servers and when a sufficient number of servers suspect the host, it is declared faulty. The sufficient number in this case is $M + 1$, where M is the maximum number of faulty hosts. Thus, even if the M faulty hosts have a receive omission or early timing failure and suspect some non-faulty host X, the non-faulty host X will not be removed from the membership since it will not be suspected by $M+1$ hosts. Naturally, since the M faulty hosts may also be crash faulty, M must be small enough to ensure there will be $M+1$ votes from correct hosts. Thus, M must be less than $N/2$.

Our design does not provide a predefined algorithm for detecting value failures. This is because the intended uses of the GRPC may vary, from accessing completely identical replicated servers to parallel access of non-identical servers. Also, the data types of the requests and responses are naturally application specific. However, our design allows the client application to specify a voting function that is used to combine the responses. The voting function may employ techniques such as majority voting, range checks, and other application-specific sanity checks. The voting function may also raise events to notify the composite protocol about server failures.

Byzantine failures require a voting protocol based on Byzantine agreement. Since we use authenticated messages, no additional redundancy is required compared with receive omission and early timing failures. That is, $M + 1$ votes are required, where M is less than $N/2$.

3.3.3. Service property algorithms. The implementation of the basic GRPC properties is all based on well known algorithms. Synchronous calls are implemented by blocking the client thread on a private semaphore until the call is completed. Note, however, that the decision on when a call is completed depends on the failure model. Asynchronous calls return immediately, and the eventual result is delivered to the client using an upcall. Unique execution is based on identifying each request with a (client-id, sequence number) pair, except in the Byzantine case, where a client-id, sequence number, and the actual data bytes are used to uniquely identify a client request. The latter is necessary to guard against a faulty client sending different byte sequences with the same client-id and sequence number to different servers. Finally, FIFO message ordering is implemented by each server queuing client requests until all calls with lower sequence numbers from the same client are received. If an expected call is not received within a specified period of time, the client is declared faulty. In this case, all calls from that client are removed from the queue.

The atomicity algorithm in the non-Byzantine case is flexible and allows customization of the tradeoff between message overhead and the worst-case latency of the GRPC request. In our approach, each server maintains a *server list* (SL), with a pointer to indicate the next server to which message information will be sent. Each server also maintains an *identifier list* (IL) containing the unique identifiers for the requests that it has received. When a new request is received, its identifier is added to the IL and the list is sent to the next k servers in the SL. The pointer in the SL is then incremented by k. On receiving the IL, a server checks if it has received all the requests it contains; if not, it asks the sender to forward the missing requests. A server removes a request from its IL after a list containing the request has been sent to all servers at least once.

The tradeoff between overhead and latency is determined by the value chosen for k. If $k = 1$, each server transmits one extra message and thus, the cost of atomicity is $O(N)$. In contrast, if $k = N - 1$, each server multicasts the IL at a cost of $O(N^2)$, but servers are able to detect and retrieve missing messages more quickly. Naturally, the cost of atomicity can be reduced further by sending the list to the next k servers only after some number j of new requests have been received.

The algorithm for total order in the non-Byzantine case builds on the atomicity algorithm. One of the servers is elected as a coordinator to order all requests. The order is propagated to all the servers in the IL messages: i.e., for each request, the list contains the unique identifier and the specified order. Since a request cannot be processed before the order is received, the standard slowly propagating atomicity algorithm would cause too much delay. Thus, the coordinator always sends the IL message to all the servers. A coordinator failure is handled by a multiphase algorithm in which the process with the next highest IP number becomes the new coordinator. It collects information from all other servers to determine the last request ordered by the previous coordinator, and then resumes normal processing.

The above total ordering algorithm does not guarantee FIFO ordering. Thus, if FIFO and total order are both required, an additional FIFO algorithm is used. The only difference with the regular FIFO algorithm is that the FIFO queue now contains totally ordered requests.

The algorithms used for Byzantine failures are based on Byzantine agreement with signed messages [4], in which multiple rounds of message exchange are used. In each round, each server sends a message to all the other servers containing a set of requests. The specific algorithm involves grouping consecutive rounds together into a *block*, which is defined as $M + 1$ rounds. In each round, each server signs and forwards the requests it receives to all other servers. Any request received from a client during a block is queued and handled in the next block. When a server receives a forwarded request from another server, it signs and forwards it to all servers that have not signed the request. After $M + 1$ rounds,

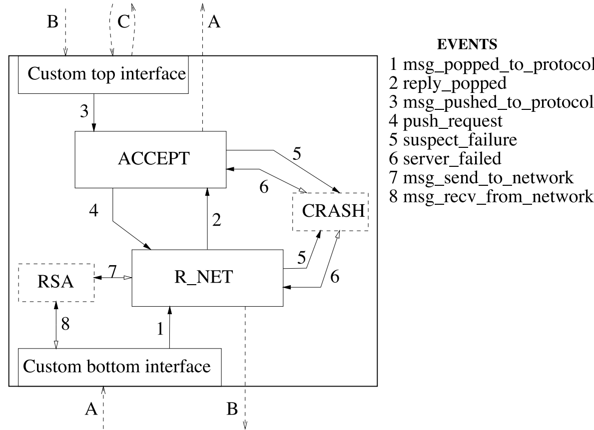

Figure 1. Client micro-protocols.

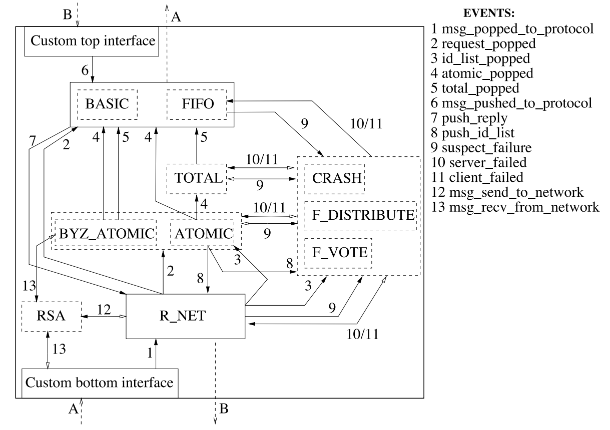

Figure 2. Server micro-protocols.

at least one non-faulty server has seen the message and hence, forwarded it to all servers. Then, the processing of the next block is started. If included, a total order micro-protocol orders the requests in each block after it is completed. Since all requests in a block are seen by all servers by the end of the block, a local deterministic ordering is sufficient.

For every set of requests, there is an exchange of messages between every pair of servers, meaning that the algorithm uses O(N^2) messages. The latency may be high, however, since every request has to wait for an entire block to be processed before being serviced by any server. Also, RSA signing and verification take considerable time, adding to the latency.

4. Implementation

The implementation of the configurable GRPC uses Cactus, where micro-protocols are used to implement each of the different logical algorithms described in section 3.3.

4.1. Client micro-protocols

Some micro-protocols are used on both clients and servers, but the set on the client is smaller and, thus, simpler as a starting point. Figure 1 represents these micro-protocols and how they interact using Cactus events. Compulsory micro-protocols are represented by solid boxes and optional ones with dashed boxes. The numbered arrows represent events, with single-headed arrows representing non-blocking events and double-headed arrows representing blocking events. In either case, the micro-protocols pointed to by the solid head service the event, i.e. they have an event handler bound for the event. Finally, A, B, and C denote interactions between CORDS protocols, with A being a message pop, B being a message push, and C being a synchronous call. Note that the custom interfaces at the top and bottom of the composite protocol translate the interaction with other CORDS protocols into events.

As indicated by the figure, the client has only two compulsory and two optional micro-protocols. The R_NET micro-protocol implements the reliable communication using a sliding window protocol. It unpacks the message headers and removes piggybacked acknowledgments before forwarding the message to other micro-protocols using appropriate events. If it suspects a failure of a server based on a delayed or missing response, it raises a failure event. If a server failure is declared, the micro-protocol stops accepting its messages. The ACCEPT micro-protocol implements the synchronous and asynchronous call variations. It sends the client request to all servers and collects responses until a required number is received. At this point, the response is returned to the client. The micro-protocol keeps track of functioning servers, so that it knows when every non-faulty server has responded and thus, the call is completed.

The optional CRASH micro-protocol waits for suspect_failure events raised by other micro-protocols, and raises either a server_failed event or client_failed event depending on the role of the failed site. The optional RSA micro-protocol signs messages and verifies message signatures using the RSA algorithm.

4.2. Server micro-protocols

The server micro-protocols and their interactions are presented in figure 2. Some of the micro-protocols—R_NET, RSA, and CRASH—and events are identical to those on the client, but a number of new micro-protocols and events are introduced.

The figure indicates that each configuration of the service must have either BASIC or FIFO micro-protocols. The BASIC micro-protocol simply forwards requests to the application level server, matches responses to requests, and returns responses to the client through the R_NET micro-protocol. The FIFO micro-protocol implements FIFO ordering on top of totally ordered or non-ordered messages, maintaining the necessary queues and other data structures as described in section 3.3. It also raises the suspect_failure event if a missing message is not received within a specified period of time, and has handlers for the client and server failure events so that it can clear any data structures for the failed hosts. Note that FIFO only requires reliability, i.e. the presence of R_NET in the configuration. When ATOMIC or TOTAL is included in addition to FIFO, they simply intercept the event request_popped, and FIFO will get the message to be ordered through either event atomic_popped or total_popped. Due to the flexible event mechanisms, FIFO actually uses one event handler to handle all three different events.

The optional ATOMIC, TOTAL, and BYZ_ATOMIC micro-protocols implement atomicity and total ordering for non-Byzantine and Byzantine failures using the algorithms described in section 3.3. In particular, the ATOMIC micro-protocol maintains the list of request identifiers and sends it out at the desired frequency. It also requests retransmissions if it detects that a message is missing based on a request list it received, and handles retransmission requests from other servers. The TOTAL micro-protocol implements coordinator-based total ordering, including dealing with coordinator failures. It relies on ATOMIC to ensure that all servers will receive the same set of requests. The details are omitted here for brevity.

The BYZ_ATOMIC micro-protocol implements the basic Byzantine agreement algorithm using authentication, collecting requests in a queue to wait for the beginning of the block in which this set of requests are processed. Note that all requests forwarded by other servers must be checked for authenticity—each forwarded request is signed by all the servers that have forwarded the request. In every round, a server forwards each of the messages it received in the previous round to all servers that have not yet signed the message.

Interactions between micro-protocols are implemented using events. If any of ATOMIC, TOTAL, or BYZ_ATOMIC are included in a configuration, they have their event handlers bound to the event request_popped before the handlers of BASIC or FIFO, so that they can stop the event and prevent BASIC or FIFO from seeing these messages. Later, when all the required properties are satisfied for the messages, these micro-protocols notify BASIC or FIFO using different events. However, if none of these three are in a configuration, the request_popped event will notify BASIC or FIFO, which will then deliver the request to the application-level server.

The server micro-protocol suite includes two more optional failure-handling micro-protocols, F_DISTRIBUTE and F_VOTE. F_DISTRIBUTE handles send omission and late timing failures. It converts local failure detections—i.e. the suspect_failure event—into an agreed membership change event server_failed or client_failed. It also distributes the local failure detection information by piggybacking it on the request identifier messages. The piggybacking is done when push_id_list is raised. It also extracts any failure information from the request identifier messages at the receiver and raises the appropriate failure events.

F_VOTE implements the voting required for receive omission, early timing, and Byzantine failures by also piggybacking the local suspicion information and raising the appropriate events when at least $M + 1$ hosts suspect a failure. Note that in the Byzantine case, the suspicion information is piggybacked with the client request messages in the agreement protocol. As a result, no host can send conflicting detection information to other hosts undetected.

4.3. Performance

The configurable GRPC service allows a tradeoff between the number of failures that can be tolerated and the service response time. While crash failures are relatively inexpensive to tolerate, the cost of fault tolerance increases as the more inclusive failure models are used. To measure this relative cost, as well as the cost of the various service properties, we tested the service on a cluster of Pentium PCs running the MK 7.3 operating system and connected by a 10 Mb Ethernet. As a reference point, the UDP and IP roundtrip times on this platform are around 1.15 ms and 1.12 ms, respectively.

Table 1 presents execution times for representative configurations of the GRPC service. These values are average response times in milliseconds from tests with clients executing synchronous RPC calls on a server group consisting of one to three servers depending on the failure model. The number of servers was chosen so that one failure of the given type could be tolerated. The average response time for one GRPC call is calculated by dividing the total time required to make 1000 consecutive synchronous RPC calls by 1000, except in the Byzantine case where the average was calculated for ten requests. Each configuration includes the R_NET micro-protocol and either BASIC or FIFO micro-protocols. Therefore, every configuration that does not mention the FIFO property has the BASIC micro-protocol. Total ordering is either provided by a combination of TOTAL and ATOMIC or BYZ_ATOMIC for the Byzantine case.

The most important aspect of this table is the relative costs. A number of observations, most of which are not surprising considering the algorithms used, can be made from these numbers. The response time naturally increases with the number of clients because of the extra load on the servers. As far as properties are concerned, the cost of adding FIFO is insignificant. On the other hand, atomicity and total ordering, which builds on atomicity, increase the response time considerably. This is to be expected because FIFO does not introduce extra messages, while atomicity and total order require a message exchange between servers. Those cases where adding an extra property appears to reduce response time by a small amount are attributable to experimental variation rather than any underlying differences.

As far as the failure models are concerned, the tests indicate a definite increase in response time as the complexity of the failure model increases. In the case of crash, omission, and timing failures, this is mostly due to the increased number of server replicas required to tolerate the failures. The cost difference is more substantial when considering complex properties such as atomicity and total order. This follows because these properties require communication between servers and as a result, are more sensitive to the number of servers required. Note also that these numbers only reflect normal operation, not the overhead incurred when dealing with failures. In these cases, the cost of the failure detection or suspicion mechanism is the same regardless of the failure model, while the cost of the agreement or voting algorithm will typically vary.

The cost of tolerating Byzantine failures is large compared with any of the other failure models. The cost is partially due to the overhead of signing messages using RSA and partially due to the cost incurred by multiple rounds of message passing. We used an existing software implementation of RSA based on 512-bit keys. The None and FIFO columns only include the cost of authentication, since if atomicity is not required, the Byzantine agreement protocol is not executed. The Atomic and Total columns

Table 1. Average response time (in ms).

Failure model	Clients	Servers	Properties					
			None	FIFO	Atomic	FIFO + atomic	Total	Total + FIFO
None	1	1	3.3	3.3	3.5	3.5	3.6	3.6
	2	1	5.1	5.1	5.6	5.6	5.9	6.0
Crash	1	2	4.2	4.2	5.7	5.6	6.2	6.4
	2	2	5.1	5.1	10.9	10.4	13.4	13.0
Send omission or late timing	1	2	4.2	4.1	5.8	5.7	6.6	6.5
	2	2	5.1	5.0	10.7	11.0	13.8	13.5
Receive omission or early timing	1	3	5.2	5.3	7.1	7.2	10.5	10.2
Byzantine	1	3	1993	2181	18 923	18 923	18 924	18 924

include the rounds algorithms and since the goal is to tolerate a single failure, the algorithm requires two rounds. The rounds algorithm is synchronous: that is, the second round only starts after the first round is completed. A timeout is used to terminate a round to ensure progress if all servers do not reply. In this test, the timeout was set to 8000 ms. This number is determined by estimating how long it takes for a host to complete a round, including encrypting and sending its set of messages and decrypting all the sets of messages it receives. If the round timeout is set too small, hosts may unnecessarily be suspected of failures.

We also measured the corresponding times for asynchronous calls. They were consistently lower—up to 30% less—than the corresponding synchronous call times. This is because asynchronous calls allow a client to issue a number of concurrent RPC calls and thus, increase the overlap between computation and communication. When the number of clients increases, however, the servers become saturated and the benefit of asynchronous calls is considerably reduced.

5. Related work

Related work on fault-tolerant distributed services is extensive. Most implementations of distributed services such as atomic multicast, membership, GRPC, synchronized clocks, and transactions tolerate at least crash failures, and some tolerate more complex failures. Typically, however, each service can only tolerate a single class of failures and is not configurable. Moreover, each implementation typically provides a different API, and relies on a specific hardware and software configuration.

Closer to the idea of customizing failure models is the family of group multicast protocols in [3], which has one protocol each for crash, omission, timing, and arbitrary failures. Another example of such a family is the set of group multicast protocols described in [19] that adapt to crash, omission, and arbitrary failures, respectively. These protocols are not configurable in the same sense as our approach, however. In particular, rather than customizing one protocol, the user must select from a related collection of protocols.

Other work dealing with multiple failure models is that on hybrid fault models [6, 20, 21]. The basic idea is to detect a range of different types of failures at the same time by using multiple failure detection techniques. This approach allows the system to tolerate a larger number of failures than traditional Byzantine algorithms, since simpler failures that require less redundancy can be distinguished from true Byzantine failures. Some algorithms designed for hybrid fault models, in particular [21], allow specification of the maximum number of faults to be tolerated for each different fault model. Although such an algorithm could exhibit comparable flexibility to our approach with respect to choice of failure model, our approach allows greater optimization of the algorithm used in each situation, since each failure model has a micro-protocol of its own.

A number of object-oriented systems support customization of fault tolerance by allowing techniques such as replication, checkpointing, and checksum error detection to be configured into an object using reflection [11, 12, 22, 23]. Although these systems support customization of fault tolerance in terms of techniques, they do not directly address the issues of customizing the failure model to be tolerated. Indeed, most existing work in this area appears to focus only on tolerating crash failures.

Other work on GRPC services has focused on customizing service properties other than fault tolerance. The event-driven model described in this paper has been used to construct GRPC services with customizable service properties such as ordering, uniqueness, orphan handling, and bounded termination [24, 25], but no support for customization of the failure model. An approach for RPC customization based on specification languages is presented in [26]. Finally, the x-kernel has been used to construct highly modular, but not configurable, RPC services [27].

6. Conclusions

The choice of a failure model is a key factor in determining the performance of a system, as well as its fault coverage. The approach presented in this paper makes it easy to construct customized services that can tolerate a given class of failures. It is based on separating the implementation of service properties and fault tolerance guarantees to the greatest extent possible using fine-grain micro-protocols and the event-based programming style in the Cactus system. We demonstrated the viability of the approach by presenting the design and implementation of a group RPC service that

supports multiple classes of failures. Performance results verify the cost incurred by using a more inclusive failure model, even when failures do not occur.

Future work will concentrate in a number of areas. One is applying this approach to other distributed services, such as a secure communication service [28] and a distributed file system. Another is investigating techniques to reduce the cost of fault tolerance by using runtime adaptation to dynamically change the class of failures being tolerated [19]. Finally, we also intend to analyse other quality of service tradeoffs that arise in distributed computing systems designed to provide fault-tolerance, real-time [8], and security guarantees.

Acknowledgments

We would like to thank Gary Wong for his excellent comments and suggestions that improved the paper. We would also like to thank the anonymous referees for their insightful comments. This work was supported in part by the Defense Advanced Research Projects Agency under Grant N66001-97-C-8518, the Office of Naval Research under Grant N00014-96-0207, and the National Science Foundation under Grant CDA-9500991.

References

[1] Schneider F 1990 Implementing fault-tolerant services using the state machine approach: a tutorial *ACM Comput. Surv.* **22** 299–319

[2] Chang J and Maxemchuk N 1984 Reliable broadcast protocols. *ACM Trans. Comput. Syst.* **2** 251–73

[3] Cristian F, Aghili H, Strong R and Dolev D 1985 Atomic broadcast: from simple message diffusion to Byzantine agreement *Proc. 15th Symp. on Fault-Tolerant Computing (Ann Arbor, MI)* pp 200–6

[4] Lamport L, Shostak R and Pease M 1982 The Byzantine generals problem *ACM Trans. Program. Lang. Syst.* **4** 382–401

[5] Powell D 1992 Failure mode assumptions and assumption coverage *Proc. 22nd IEEE Symp. on Fault-Tolerant Computing (Boston, MA)* pp 386–95

[6] Thambidurai P and Park Y-K 1988 Interactive consistency with multiple failure modes *Proc. 7th Symp. on Reliable Distributed Systems* (Los Alamitos, CA: IEEE Computer Society Press) pp 93–100

[7] Bhatti N, Hiltunen M, Schlichting R and Chiu W 1998 Coyote: a system for constructing fine-grain configurable communication services *ACM Trans. Comput. Syst.* **16** 321–66

[8] Hiltunen M, Schlichting R, Han X, Cardozo M and Das R 1999 Real-time dependable channels: customizing QoS attributes for distributed systems *IEEE Trans. Parallel Distrib. Syst.* **10** 600–12

[9] Travostino F, Menze E and Reynolds F 1996 Paths: programming with system resources in support of real-time distributed applications *Proc. IEEE Workshop on Object-Oriented Real-Time Dependable Systems (Laguna Beach, CA)* (Los Alamitos, CA: IEEE Computer Society Press)

[10] Hutchinson N and Peterson L 1991 The *x*-kernel: an architecture for implementing network protocols *IEEE Trans. Software Eng.* **17** 64–76

[11] Agha G and Sturman D 1994 A methodology for adapting to patterns of faults *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems* ed G Koob and C Lau (Dordrecht: Kluwer) pp 23–60

[12] Fabre J-C and Perennou T 1998 A metaobject architecture for fault tolerant distributed systems: the FRIENDS approach *IEEE Trans. Comput. (Special Issue)* **47** 78–95

[13] Nelson B 1981 Remote procedure call *PhD Thesis* Department of Computer Science (Pittsburgh, PA: Carnegie-Mellon University)

[14] Cheriton D 1986 VMTP: a transport protocol for the next generation of communication systems *SIGCOMM'86: Proc. Symp. on Communication Architectures and Protocols* (New York: ACM Press) pp 406–15

[15] Birman K 1985 Replication and fault-tolerance in the Isis system *Proc. 10th ACM Symp. Operating System Principles (Orcas Island, WA)* pp 79–86

[16] Kopetz H, Grunsteidl G and Reisinger J 1991 Fault-tolerant membership service in a synchronous distributed real-time system *Dependable Computing for Critical Applications* ed A Avizienis and J C Laprie (Vienna: Springer) pp 411–29

[17] Bernstein P, Hadzilacos V and Goodman N 1987 *Concurrency Control and Recovery in Database Systems* (New York: Addison-Wesley)

[18] Fischer M, Lynch N and Paterson M 1985 Impossibility of distributed consensus with one faulty process *J. ACM* **32** 374–82

[19] Chang I, Hiltunen M and Schlichting R 1998 Affordable fault tolerance through adaptation *Parallel and Distributed Processing (Lecture Notes in Computer Science vol 1388)* ed J Rolin (New York: Springer) pp 585–603

[20] Lincoln P and Rushby J 1993 A formally verified algorithm for interactive consistency under a hybrid fault model *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (Toulouse)* pp 402–11

[21] Walter C, Hugue M and Suri N 1995 Continual on-line diagnosis of hybrid faults *Dependable Computing for Critical Applications* vol 4, ed F Cristian *et al* (Vienna: Springer) pp 233–49

[22] Sturman D and Agha G 1994 A protocol description language for customizing failure semantics *Proc. 13th Symp. on Reliable Distributed Systems* (Los Alamitos, CA: IEEE Computer Society Press) pp 148–57

[23] Fabre J-C, Nicomette V, Perennou T, Stroud R and Wu Z 1995 Implementing fault tolerant applications using reflective object-oriented programming *Proc. 25th Int. Symp. on Fault-Tolerant Computing (Pasadena, CA)* pp 489–98

[24] Hiltunen M and Schlichting R 1995 Constructing a configurable group RPC service *Proc. 15th Int. Conf. on Distributed Computing Systems (Vancouver)* pp 288–95

[25] Bhatti N and Schlichting R 1995 A system for constructing configurable high-level protocols *Proc. SIGCOMM '95 (Cambridge, MA)* pp 138–50

[26] Huang Y-M and Ravishankar C 1994 Designing an agent synthesis system for cross-RPC communication *IEEE Trans. Software Eng.* **19** 188–98

[27] Hutchinson N, Peterson L, O'Malley S and Abbott M 1989 RPC in the *x*-kernel: evaluating new design techniques *Proc. 12th ACM Symp. on Operating Systems Principles (Litchfield Park, AZ)* pp 91–101

[28] Hiltunen M, Jaiprakash S and Schlichting R 1999 Exploiting fine-grain configurability for secure communication *Technical Report 99-08* University of Arizona