School of Computing Science,
University of Newcastle upon Tyne

From Crash Tolerance to Authenticated Byzantine Tolerance: a Structured Approach, the Cost and Benefits

Mpoeleng, D., Ezhilchelvan, P.D. and Speirs, N.A.

Technical Report Series

CS-TR-824

January 2004

From Crash Tolerance to Authenticated Byzantine Tolerance: A Structured Approach, the Cost and Benefits

Dimane Mpoeleng, Paul Ezhilchelvan and Neil Speirs
School of Computing Science
University of Newcastle upon Tyne NE1 7RU, UK
{dimane.mpoeleng, paul.ezhilchelvan, neil.speirs}@ncl.ac.uk

Abstract

Many fault-tolerant group communication middleware systems have been implemented assuming crash failure semantics. While this assumption is not unreasonable, it becomes hard to justify when applications are required to meet high reliability requirements and are built using commercial off the shelf (COTS) components. This paper presents a structured approach to extend a crash-tolerant middleware system into an authenticated Byzantine tolerant one with small modifications to the original system. The proposed approach is based on state machine replication (SMR) and is motivated by the composability features of standard distributed object technologies such as CORBA. SMR is used to assure signal-on-failure (fail-signal) semantics at a level where existing crash-tolerant services can be seamlessly deployed. The resulting system can provide total ordering that has no liveness requirement for termination. We demonstrate the effectiveness of our approach by porting a crash-tolerant CORBA group communication service – the NewTOP system. We also measure the performance of the resulting system.

Keywords and phrases*: Authenticated Byzantine failures, State machine replication, self-checking, fail- signal, total order, CORBA, group communication*

1. Introduction

We address the problem of building a Byzantine fault tolerant, group-communication middleware system for asynchronous communication networks. Such a system offers services that simplify the development of distributed applications over an asynchronous network (e.g., the Internet) which supports operational processes to exchange messages but guarantees no bound on message delays. The services include: reliable multicast, causal order and total order (or atomic multicast). The last one is essential for replicating application servers and providing fault-tolerant services. It is also harder to achieve and the difficulties are epitomized in the FLP impossibility result [FLP85]: there can be no deterministic total order protocol if processes can crash.

Crash-tolerant middleware systems deal with the FLP impossibility in one of three ways. In *partitionable* systems (e.g., [CF99, DM96, EMS95]), middleware processes that do not suspect each other, remove from the group those processes which they suspect to have crashed. Since the bound on message delays is not known precisely, suspicions can be false; this can lead to connected, operational processes being split into sub-groups (logical partitions) even when no process has crashed. Group-splitting thus reduces the fault-tolerance potentials of a group; merging the partitioned sub-groups and ensuring state reconciliation is a hard problem and an automated solution typically requires considerable message and time overhead [LKD97].

This problem does not exist in a *non-partitionable* system such as [FGS98]. However, termination of a deterministic total order protocol is guaranteed, even in failure-free runs, only when message delays over the asynchronous network are perceived to remain stable for a suitably long duration. We refer the reader to [CT96] which precisely states this requirement in its weakest form as *w*. If, on the other hand, non-deterministic or randomized protocols are used, termination requires that the random choices made converge and this is guaranteed, only in probabilistic terms,

to be a certainty with the passage of time [EMR01]. Such requirements for termination are often called the *liveness* requirements. They make it hard to predict performance and to provide meaningful performance guarantees to applications: total-ordering latency is influenced by how early the *liveness* requirement is met during the protocol execution which, in turn, depends on the choice of values assigned to the protocol parameters. For example, in a w-based system, when timeouts chosen for suspecting failures become small compared to actual message delays, it postpones the realization of w and increases the latency even in failure-free runs; setting long timeouts, on the other hand, slows down failure detection and affects the performance when failures do occur.

For a large class of Internet-based dependable applications (e.g., e-auctions, B2B applications etc.,), an asynchronous middleware system must be robust and responsive in the following sense: (i) it must tolerate faults more serious than crashes (*robustness*), and (ii) its performance should be free of *liveness* requirements that need to be met by making appropriate choice of values either speculatively (as in timeouts) or randomly (*responsiveness*). Systems or architectures, such as [KMM98, MR98, CL99], which tolerate (authenticated) Byzantine faults do exist. (An authenticated Byzantine fault causes a component to fail in arbitrary manner that is however restricted by the effectiveness of message signature and authentication mechanisms such as the RSA scheme.) These systems make use of Byzantine fault tolerant protocols developed almost 'from scratch'. Baldoni et. al., derives a Byzantine protocol from a crash-tolerant one [BHR00]. Such protocols require at least one extra communication round than their crash-tolerant counterparts (if the latter exist) and at least $3f+1$ nodes to guarantee total order if f is the maximum number of nodes that can become faulty. They however deal with the FLP impossibility by one of the two ways attributed above to the non-partitionable approach.

We achieve the objective of robustness by considering authenticated Byzantine faults and responsiveness by seeking an alternative way to deal with the FLP impossibility. On the latter point, we observe that (a) the FLP impossibility applies only to crash model (unannounced stopping) and not to announced crashes, and (b) a fail-stop process [SS83, S84] which has been built with internal redundancy to tolerate authenticated Byzantine faults, can be guaranteed to announce its crash to its environment. These observations form the rationale for our structured approach: we first build every middleware process as a pair of self-checking processes on distinct nodes, and then construct a middleware system out of such middleware processes, termed as the *fail-signal* processes, whose failure modes are as follows.

A fail-signal process fails only by outputting fail-signals that are unique to that process. More precisely, a faulty fail-signal process

(*fs1*) outputs its fail-signal whenever it cannot produce a correct response, and

(*fs2*) may also output its fail-signal at arbitrary timing instances.

Two important remarks are in order. ***Remark 1***: By *fs1*, whenever a response is expected of a fail-signal (FS hereafter) process, it is produced; it is correct if it is not a fail-signal. Note however that the outputting of a fail-signal does not necessarily mean that a response from the signaling process was expected nor the process has crashed (*fs2*). Thus, a faulty FS process, say p_j, behaves like a correct process whose responses pass through an adversary who is restricted only to substituting an arbitrary subset of p_j's responses with p_j's fail-signals or to randomly emitting p_j's fail-signals. A fail-stop process [SS83] – the inspiration for our FS process – offers stronger failure-guarantees: its fail-signal is a sure indication of its crash and its pre-crash states are preserved. Because of this, a 3-fold redundancy is needed for fail-stop construction, whereas a 2-fold redundancy will suffice for constructing an FS process. (Details in Section 2.)

Remark 2: Since a signaling FS process is necessarily faulty, a process that receives a fail-signal can correctly regard the source to be faulty; i.e., failure detection does not involve choosing appropriate timeouts which cannot always be done correctly over an asynchronous network. Thus, with the FS middleware processes, the FLP impossibility result ceases to hold and it is possible to have a *deterministic* total order protocol that neither tends to split groups in the absence of failures nor requires the existence of w (or a similar liveness requirement [CL99]).

One cost aspect of our approach is straightforward. Masking of f Byzantine faults at

the application level requires at least $2f+1$ replicas, with each replica requiring access to a total-order service. Since we construct our total-order service using FS middleware processes and each FS process itself is a self-checking pair running on distinct nodes, $4f+2$ nodes are needed in our approach i.e., $(f+1)$ nodes more than the known optimal requirement for a Byzantine-tolerant middleware system. We believe that this cost will be small, given the falling hardware price.

The objective of this paper is twofold: to demonstrate that our fail-signal based approach can considerably simplify the middleware construction through software re-use and to evaluate the performance degradation when crash-tolerance is swapped for (authenticated) Byzantine tolerance. When the construction of a deterministic, crash-tolerant middleware system and that of FS processes conform to the same standard, integration that leads to a Byzantine-tolerant middleware system, requires very little code change to the original crash-tolerant system. As a proof of concept we enhance the NewTOP system which is a CORBA compliant, crash-tolerant, and partitionable middleware system [MS00, ME00], with FS middleware processes. The enhanced NewTOP, called the FS-NewTOP hereafter, tolerates authenticated Byzantine faults and does not lead to group partitioning. We then measure the ordering latencies of FS-NewTOP and the original NewTOP under those conditions that put the components of FS processes under maximum processing load. The paper is organized as follows. The next section describes the construction of fail-signal (FS) processes and the assumptions involved. Section 3 describes the existing NewTOP and how the FS-NewTOP is constructed as an extension of the existing system. Section 4 presents the experimental set-up, measures the latency and throughput cost extracted by this extension. Section 5 concludes the paper.

2. Construction of Fail-Signal (FS) Processes

2.1. Assumptions and Principles

To transform a middleware process p into an FS p, p must be a deterministic state machine in the sense that the execution of an operation by p in a given state and with a given set of arguments must always produce the same result (requirement **R1**). Middleware processes that implement deterministic algorithms and protocols satisfy R1. We construct FS p by hosting a replica pair, denoted as $\{p, p'\}$, on distinct nodes as shown in figure 1. (Throughout the paper, the replica pair of an object or process x is denoted as $\{x, x'\}$.) We make the following assumptions.

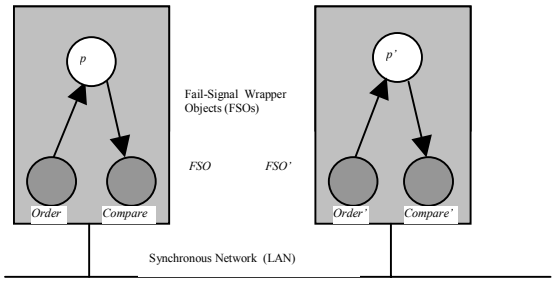

Figure 1: Architecture of Fail-Signal Wrapper Objects for middleware process p.

The nodes are assumed to be correct (i.e., non-faulty) when they are paired at start-up time and at most one of these nodes can develop faults of authenticated Byzantine type (assumption **A1**). The nodes are connected by a reliable, synchronous communication link (LAN) that delivers messages within a known bound δ (**A2**). Suppose that both nodes are non-faulty and an input is submitted to both of them at the same time t for processing. Say, p (respectively p') processes that input and generates a result at time $t+\Delta t$ (respectively at $t+\Delta t'$). We assume that $\max\{\Delta t, \Delta t'\} \leq \kappa * \min\{\Delta t, \Delta t'\}$, for some known, positive number κ (**A3**). Similarly, suppose that both nodes are non-faulty and p and p' schedule a *send_result()* operation at the same time s to forward their result to the other replica. Say, p (respectively p') completes the send operation at time $s+\Delta s$ (respectively at $s+\Delta s'$). We assume that $\max\{\Delta s, \Delta s'\} \leq \sigma * \min\{\Delta s, \Delta s'\}$, for some known, positive number σ (**A4**). A3 and A4 require that the maximum difference between the delays for processing and scheduling of middleware messages, be bounded and known. Finally, a process of a correct node can sign the messages it sends and the signed message cannot be generated nor undetectably altered by a process in another node (**A5**).

The pair $\{p, p'\}$ is made to act as a self-checking pair by means of process pairs {*Order,*

Order'} and {*Compare, Compare'}* which are threads (like *p* or *p'*) within a Fail-signal wrapper Object pair {*FSO*, *FSO'*} (see figure 1). A message destined for FS p must be received by both the wrapper objects *FSO* and *FSO'*. The Order process pair ensures that the inputs are submitted to *p* or *p'* in an identical order. The Compare processes check if *p* and *p'* generate identical outputs. If so, the output is transmitted to the destination(s), together with verifiable evidence (see below) that output checking has been carried out. Note that if a destination is an FS process, then each Compare process transmits the output to both the replicas of the destination FS process.

When *Compare* (of *FSO)* receives an output generated by *p*, it signs it and forwards a copy to *Compare'*. If it receives a signed message of identical contents from *Compare'* within a certain timeout, it signs the received message and outputs the double-signed message which will be regarded as an output of FS p. Similarly, *Compare'* will output a double-signed message where the first-signature is by *Compare*. An output from FS p is *valid* only if it bears the authentic signatures of both *Compare* and *Compare'*.

At the start-up time, when both nodes are correct, each Compare process is supplied with a fail-signal message signed by the other Compare process. When *Compare* decides that an output produced by *p* could not be successfully compared within the timeout, it signs the fail-signal supplied to it at the start-up time and emits the double-signed fail-signal to the destination(s) of that output; from this moment on, it ceases to exchange messages with the remote *Compare'* and instead it sends the double-signed fail-signal to destination(s) of any locally produced output; it also replies to the sender of any incoming message with the double-signed fail-signal. That is, *Compare* of a correct node, after detecting a failure in the other node, sends a (double-signed) fail-signal to all entities that are expecting a response from the FS p.

Observe that when both nodes are correct, two valid outputs are generated – both having identical contents and been signed by both *Compare* and *Compare'* but in different order. When faulty *p'* generates no, late, or incorrect output, *Compare* starts emitting double-signed fail-signal to destinations that expect a response from the FS p; further, *Compare* stops its interacting with *Compare'*, leaving the latter unable to produce any valid output. Thus, an FS p can fail only by emitting a fail-signal that can be uniquely attributed to it. It is possible that a node fault can cause the local Compare process to emit fail-signals arbitrarily. This leads to *fs2* described earlier.

2.2. Implementation Details

The details of implementing fail-signal processes are very similar to our earlier implementation of fail-silence processes. In the latter, a Compare process simply stops functioning when matching of output messages does not succeed. They are not therefore equipped with the single-signed fail-signal messages at the start-up time. The details of fail-silence implementation can be seen in [BESST96, BLS98] and fault-injection testing in [SSKXBI01]. For this paper, we worked on our fail-silence implementation to augment fail-signaling feature and to make the system CORBA compliant. For space reasons, we have left the details to the Appendix, except to outline some aspects of interest.

The input messages are ordered using a simple, asymmetric protocol that does not require nodes' clocks to be synchronized. One of the wrapper objects, say *FSO*, is fixed as the Leader and the other, *FSO*', as the Follower. The Order process of *FSO'*, *Order'*, accepts the order decided by *Order* and checks whether every message it receives is being ordered by the leader. This means that a given input is submitted to *p* and then to *p'*, and the time difference can be at most δ.

A Compare process computes, for every locally produced output, the time elapsed since the corresponding input was submitted for processing (as π) and the time taken to sign and forward the output to its remote counterpart (as τ). While waiting for the matching, single-signed output to be received, *Compare* uses the timeout of $2\delta + \kappa*\pi + \sigma*\tau$ and *Compare'* uses the timeout of $\delta + \kappa*\pi + \sigma*\tau$.

3. The NewTOP Group Communication Service

The Newcastle Total Order Protocol (NewTOP) is a CORBA compliant, crash-tolerant, partitionable middleware system. The system is implemented as a CORBA object called the

NewTOP Service Object (NSO). When application processes want to form a group with a common goal and to avail themselves of group communication services to this end, each process is allocated an NSO as shown in figure 2. An application process A_i acts as a 'client' to its NSO in obtaining group communication services from the latter. The communication between A_i and its NSO, and the communication between NSO's themselves are handled by an ORB.

Note that an NSO and its application 'client' need *not* reside on the same host, for the reasons that NSO is a CORBA object and the communication between an NSO and its client is handled by the ORB (*location independence*) [OMG00]; however, for performance reasons, they are normally hosted by the same node. Further, NewTOP requires an A_i to be member of a group in which A_i intends to multicast and permits A_i to be a member of more than one group at the same time. Being a partitionable system, it does not however support merging of partitioned sub-groups.

An NSO comprises of two subsystems: Invocation service and Group Communication (GC) service. The former allows the application to specify the type of NewTOP service needed and marshals a multicast message accordingly. The latter implements protocols to provide a variety of services: symmetric total order, asymmetric total order, reliable multicast, simple (unreliable) multicast and (partitionable) group membership.

When A_i multicasts a message to the group, the message is marshaled into a generic CORBA type `any` by the Invocation service and the relevant protocol of the group communication service is invoked to deliver the message. At the delivery end, the reverse happens. The Invocation service at a destination end unmarshals the delivered message (of type `any`) and delivers it to the client application A_j.

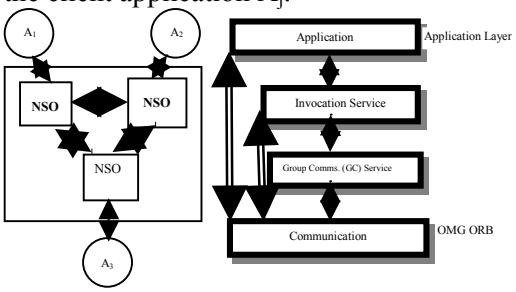

Figure 2: The NewTOP service: access and structure.

If a host node of, say, A_1 in figure 2 develops Byzantine faults, the faults can manifest at two levels. First, at the *application level*, the message which A_1 multicasts to the group may contain erroneous information. To tolerate this failure, A_2 and A_3 must be replicas of A_1; given that the latter are correct, the failure of A_1 can be masked through a majority vote. Secondly, the fault may manifest at the *middleware level*. The NSO associated with A_1, when hosted in the same node as A_1, may corrupt, probably undetectably, A_1's multicast message. NewTOP, designed to be only crash-tolerant, cannot tolerate such failures and provide correct middleware services. It is the middleware-level failures of non-benign nature that we wish to tolerate by extending NewTOP into a Byzantine tolerant one.

3.1. Extending NewTOP to FS-NewTOP

Figure 3 depicts the structure of the FS-NewTOP system, extended from (crash-tolerant) NewTOP by using an extra node, a synchronous link to connect the node pair, and the Fail Signal Wrapper Objects whose target is the NewTOP group communication (GC) service object. The wrapping of GC is made transparent to GC. To achieve this transparency, CORBA interceptors are used [NMM99]. A call to NewTOP GC, either from the Invocation layer or from a remote NewTOP GC, is intercepted on the fly and is submitted to both *GC* and *GC*' in an identical order with the *FSO* acting as the leader. Similarly, a double-signed response returned by *FSO* and *FSO'* to the Invocation layer is intercepted, signatures stripped and duplicates suppressed. This interceptor based technique used here is very similar to the one used in the Eternal system [NMM00]. Since the GC service is implemented as a single-threaded, deterministic application, *GC* and *GC*' run as deterministic state machines regardless of other software (e.g. CORBA) running on the host nodes.

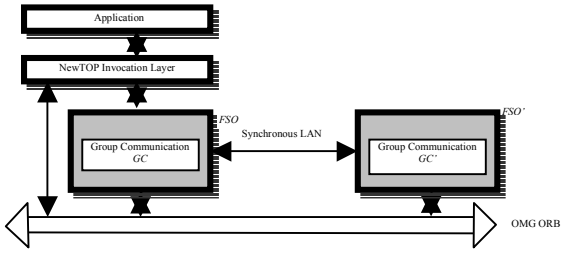

Figure 3. The FS-NewTOP system

Observe that the wrapper transparency means that *GC* and *GC'* regard themselves as the

only GC service below the Invocation layer as in the original NewTOP. Futher, that the *GC'* is hosted on a different node to the Invocation layer will not matter since the communication between the two is via the ORB (which hides location) and through the wrapper object *FSO'* which is a CORBA object. Thus, with the CORBA-compliant fail-signal wrappers and the ORB technology, the extension to FS-NewTOP was seamless. Indeed, the applications can specify, as a NewTOP service option, whether Byzantine tolerance is needed or crash tolerance is sufficient. In the latter case, *FSO* will not choose to order the input and compare the output; *FSO'* will remain unused. Adding this functionality will be a future work. Below, we state the modification to the original NewTOP necessary for the extension.

The NewTOP group membership object in GC system makes use of a failure suspector module which periodically 'pings' remote NSO GCs and generate suspicions based on a timeout mechanism. In the FS-NewTOP, a suspector module does not have to send 'pings'; instead, it converts the fail-signals received into 'suspicions' and supplies them to the group membership object. As stated earlier, fail-signals uniquely identify, and are indications of a fault at, the signaling entity; so, the suspicions generated in FS-NewTOP, unlike those in NewTOP, cannot be false. This avoids splitting of groups when there are no failures and preserves all correct FS-GCs in one group. Note also that all input messages are submitted to *GC* and *GC*' in an identical order. Therefore the suspector modules of *GC* and *GC'* send suspicions to the group membership objects of *GC* and *GC'* in an identical order. Since the NewTOP group membership protocol is deterministic, the outputs (group views) computed by the group membership objects of *GC* and *GC'* will be identical.

Referring to figure 3, a non-benign failure at the Invocation layer that results in an application message being lost or corrupted can be treated as an application-level non-benign failure, mentioned earlier. Total order protocols are unconcerned with the correctness of the application-level contents of the messages they order. So, even if NewTOP-GC were to implement a *w* based (crash-tolerant) protocol, by the fail-signaling properties of FS-GCs, the requirements of *w* will be met so long as FS middleware processes are not permanently disconnected and a majority of them remain correct. The reader is referred to [E02] which transforms a *w* based, crash-tolerant total-order protocol for a (mixed) system of *f* FS processes and (*f+1*) Byzantine-prone processes.

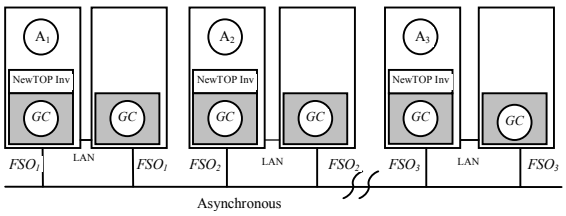

Figure 4: Deployment of the components of FS-NewTOP for a 3-Member Group.

Figure 4 shows the deployment of FS-NewTOP for a 3-member application group $\{A_1, A_2, A_3\}$. For a given *i,* $1 \leq i \leq 3$, FSO_i and FSO'_i are placed in nodes connected by a synchronous LAN; they are also connected to every $\{FSO_i, FSO'_j\}$, $i \neq j$, by the (reliable) asynchronous network. At most one node fault (of authenticated Byzantine nature) can be tolerated by the system shown in figure 4. If the node of an FSO'_i is faulty, A_i cannot effectively communicate with the rest of the group due to FSO_i having stopped the middleware operations or fail-signals being emitted arbitrarily (by FSO'_i). If the node of an FSO_i is faulty, the following failure mode is also possible: A_i can transmit messages of application-specific erroneous contents. If A_i's are replicas of each other, a client of this replica group must multicast its request to the entire group and must majority-vote the results received from the replicas. Thus, with the FS-NewTOP, $4f+2$ nodes are needed to mask *f* Byzantine faults. The other cost of FS-NewTOP over NewTOP is the performance degradation due to fail-signaling which is measured in the next section.

4. Performance Cost of NewTOP Extension

We have run a set of experiments to evaluate the performance degradation due to providing the fail signal guarantee to NewTOP middleware processes. The experiment set-up was chosen to evaluate the *maximum* degradation. This was achieved in two ways. Among the NewTOP services, the symmetric total order protocol is known to be significantly message intensive

[ME00, MS00]. (It orders a message only after the message is logically acknowledged by *all* members in the group.) Its execution is expected to keep the cost of self-checking within FS-GC at its maximum. Secondly, false failure suspicions in NewTOP runs were eliminated, as they can lead to unnecessary group-splitting and construction of new views which can only favour FS-NewTOP. To eliminate false suspicions, node failures were disallowed and a lightly-loaded LAN was chosen (in place of an asynchronous network) so that the large timeouts used will always be larger than the actual message delays encountered. (Note that the large timeouts degrade performance only when nodes do fail.)

A corollary of the above assumptions made only for the experimental setup means that a node that hosts A_i can be made to host two wrapper objects $\{FSO_i , FSO'_j\}$, $i \neq j$, without violating assumption A2 (in section 2.1). This halved the number of nodes needed to deploy FS-NewTOP for an application group of a given size. Figure 5 shows the deployment of FS-NewTOP for a 3-member group only with three nodes (instead of 6 nodes as shown in figure 4). Since each node hosts two wrapper objects in FS-NewTOP runs compared to hosting just one GC in NewTOP runs, this arrangement again favours NewTOP in the estimation of the effect of fail-signal overhead.

Our experiments used 16 Pentium III Dual Processor PC's with 512Mbytes of memory connected by a 100mb LAN. We evaluated the overhead for groups of up to 15 members, using 15 nodes and leaving one node for the collection of performance statistics. Both the NewTOP and FS-NewTOP systems were implemented using Java 1.4.0 (which comes with its own ORB).

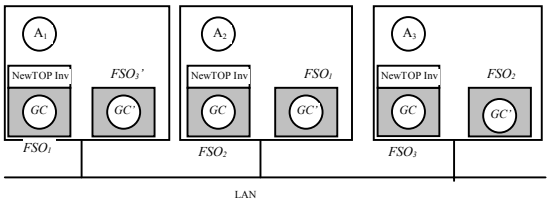

Figure 5: Placement of *FSO'* in the Experimental Set-up.

We first measured the time required to (symmetrically) total order small (3 byte) messages for groups of size between 2 and 15 members. Each A_i multicast 1000 messages for total ordering at a regular interval that was identical in both NewTOP and FS-NewTOP runs. The latency figures for FS-NewTOP and NewTOP are shown in Figure 6.

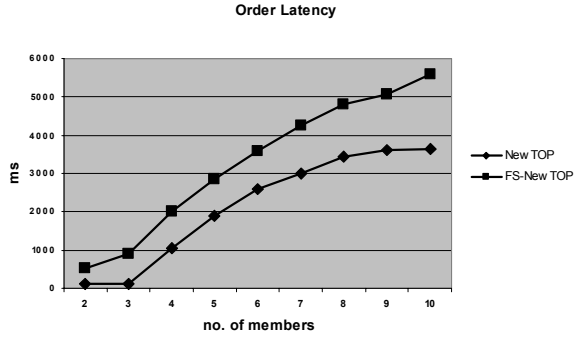

Figure 6: The Ordering Latency.

FS-NewTOP shows a fairly constant latency difference to NewTOP for small groups and the overhead is around 50% (1.5-2 seconds per message) for groups with 9 and10 members. The higher latency of FS-NewTOP comes from three sources: authenticating input messages, the leader *FSO* waiting for a matching output from the follower *FSO'* (who always lags behind the leader), and the signing of output messages (performed using the Java security package with MD5 using RSA encryption signature algorithm). As the group size increases, the message processing load on nodes increases, resulting in an increasing difference.

We then measured the throughput of the two systems by recording the time needed to order 1000 messages sent by each A_i. The results are shown in Figure 7.

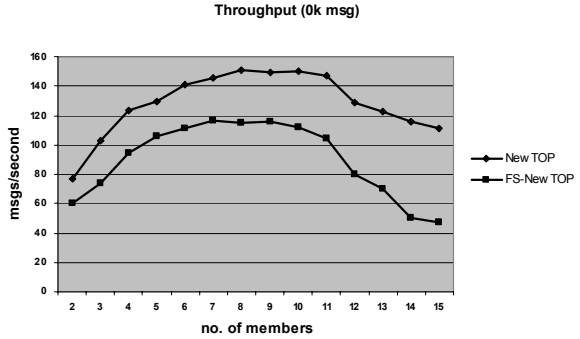

Figure 7: Throughput of NewTOP and FS-NewTOP.

An interesting observation is that both systems provide better throughput as the number of group members increase from 2. The reason for this counter-intuitive result is due to CORBA's

efficient thread handling mechanism. NewTOP and FS-NewTOP have a configurable thread pool with a default of 10 threads to handle incoming requests. Throughput drops noticeably for both systems when groups are larger than the size of the thread pool. FS-NewTOP has a throughput overhead of around 20-30% for small groups rising to 100% for groups with more than 10 members.

We then experimented by changing the size of the messages sent for a fixed group with 10 members. The throughput results are shown in Figure 8. It can be seen that the throughout of both NewTOP and FS-NewTOP decreases with increasing message size. The throughput overhead of FS-NewTOP is nearly constant at around 30 messages per second irrespective of message size.

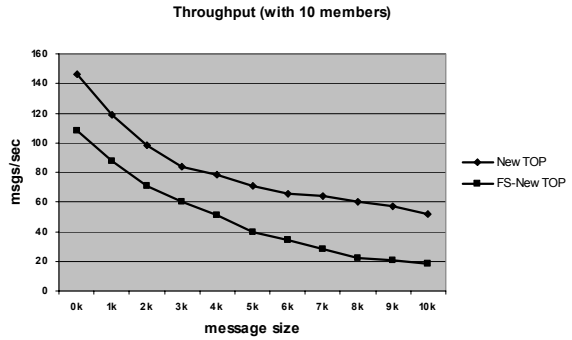

Figure 8: Throughput of NewTOP and FS-NewTOP for varying message sizes.

5. Concluding Remarks

We have constructed a Byzantine-tolerant, group-communication system by extending a crash-tolerant system. The extension involves replacing crash-prone middleware processes with fail-signal or FS processes. The idea of signal-on-failure is not new and it is one of the three failure-guarantees offered by the Fail-stop processes of [SS83]. A Fail-stop process requires (at least) three (internal) replicas, while an FS process can be done with a replica pair. A significant benefit of our fail-signal based approach is that the FLP impossibility result derived for unannounced crashes ceases to apply and consequently the total ordering is guaranteed to terminate so long as the asynchronous network does not suffer permanent partitions. The fail-signal overhead was measured through a series of experiments. The increase in ordering latency and the reduction in throughput were found to be relatively small for large groups. We intend to build the fail-signal support mechanism as an EJB container, which will enable us to experiment with upgrading of other well-known crash-tolerant middleware systems.

The assumptions made in the construction of FS processes have implications at the application and middleware levels. Assumptions A3 and A4 (in Section 2.1) require that the maximum delay within which the replicas of an FS middleware process complete processing of an incoming message or scheduling an outgoing message, must be known and bounded. Otherwise, correct replicas might find each other untimely and start emitting fail-signals unnecessarily. When the load imposed by application level processing is unknown, realizing A3 and A4 will require that the replicas be run with a high priority. We note here that an application process, such as A_i in Figure 4, can also be made into an FS A_i in the same way middleware processes were transformed. This will incur an additional FS overhead at the application level and subject replicas of A_i to assumptions A3 and A4. Alternatively, when A_i is Byzantine fault-prone (as in figure 4), another application, say B, can be replicated on the host nodes of *FSO'*. ([E02] considers such an arrangement.)

Fail-signal construction also assumes that the two nodes of an FS process are connected by a synchronous link (assumption A2) and that no more than one node becomes faulty (assumption A1). A2 can be realized, say, by keeping each pair of nodes geographically close and making use of the fast Ethernet technology. Realizing A1 in an intrusion prone environment requires sufficient diversity and intrusion detection measures, which will be the focus of our future work; in this paper, we have assumed that the causes of the faults can only be internal. Our middleware system requires a total of $4f+2$ nodes, with A1 restricting the locations of faults; in [E02], we argue that this can be reduced to the standard requirement of $3f+1$. However, the traditional Byzantine-tolerant total-order protocols neither require A1 nor anything similar to it. On the other hand, they rely on protocol-specific, *liveness* conditions to prevail for termination. In its design philosophy, [VC02] is similar to ours and assumes a synchronous WAN to avoid the FLP impossibility result. Much of the motivation for its design, expressed elegantly using the *Wormholes* metaphor [V03], also underpin our approach.

Acknowledgements: We acknowledge Graham Morgan and Doug Palmer who implemented NewTOP; also the constructive criticisms of the DSN reviewers and the careful shepherding by Matti Hiltunen; and the financial support by the EU-IST TAPAS project.

References

[BESST96]F. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. Speirs and S. Tao, "Implementing fail-silent nodes for distributed systems", *IEEE Transactions on Computers*, 45(11), pp. 1226-1238, November 1996.

[BLS98] D. Black, C. Low and S.K. Shrivastava, "The Voltan Application Programming Environment for Fail-Silent Processes", *Distributed Systems Engineering*, vol. 5, pp. 66-77, June 1998.

[BHR00] R. Baldoni, J.-M. Helary, and M. Raynal. "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach". *In Proceedings of the International Conference on Dependable Systems and Networks*, pages 273-282, New York, NY USA, June, 2000.

[C95] L. Chae, "Fast Ethernet: 100BaseT", *Network Magazine*, Dec., 1995, URL: http://www.networkmagazine.com/article/NMG20000727S0014.

[CF99] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10 (6), June 1999, pp. 642-57.

[CL99] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *In Proceedings of the 3rd ACM Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 99, pp. 173-186.

[CT96] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems*", Journal of the ACM*, 43(2), pp. 225 - 267, March 1996.

[DM96] D. Dolev and D. Malkhi, "The Transis Approach to High Availability Cluster Communication", *Communications of the ACM*, Vol. 39, No. 4, pp. 64-74, April 1996.

[E02] P D Ezhilchelvan, "A Middleware Architecture for Intrusion Tolerant Service Replication", *In Proceedings of the International Workshop on Intrusion Tolerant Systems*, pp. C-6-1 – C-6-7, In association with IEEE International Conference on Dependable Systems and Networks (DSN02), Washington, DC, June 23 - 26, 2002

[EMS95] P. Ezhilchelvan, R. Macedo and S. K. Shrivastava, "Newtop: a fault-tolerant group communication protocol", *In Proceedings of the 15th IEEE Intl. Conf. on Distributed Computing Systems*, Vancouver, May 1995, pp. 296-306.

[EMR01] P D Ezhilchelvan, A Mostefaoui, and M Raynal, "Randomized Multivalued Consensus", *In Proceedings of the Fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC),* May 2001, Magdeburg, Germany, pp. 195-201.

[FGS98] P. Felber, R. Guerraoui and A. Schiper, "The implementation of CORBA Object service", *Theory and Practice of Object Systems*, Vol. 4, No. 2, 1998, pp. 93-105.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, April 1985.

[KMM98] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication", *In Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS)*, pp. 317-26, Jan. 1998.

[LKD97] E. Y. Lotem, I. Keidar and D. Dolev. "Dynamic Voting for Consistent Primary Components", *In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC),* pp. 63-71, 1997.

[ME00] G. Morgan and P.D. Ezhilchelvan, "Policies for using Replica Groups and their effectiveness over the Internet", *Proc. 2nd International COST264 Workshop on Networked Group Communication (NGC 2000),* Palo Alto, California, 2000.

[MR98] D. Malkhi and M. Reiter, "Byzantine Quorum Systems", *Distributed Computing*, 11(4), pp.203-213, 1998.

[MS00] G. Morgan and S.K. Shrivastava, "Implementing Flexible Object Group Invocation in Networked Systems", *In Proceedings of the International Conference on Dependable Systems and Networks*, New York, June 2000.

[NMM99] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA", *IEEE Computer,* pp. 62-68, July 1999.

[NMM00] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "The Eternal System", In *Encyclopedia of Distributed Computing*, edited by P. Dasgupta and J. E. Urban, Kluwer Academic Publishers (2000).

[OMG00] OMG Technical Committee Document ptc/00-04-04, Object Management Group, March 2000.

[SS83] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", *ACM Transactions on Computer System*s, Vol. 1(3), pp. 222-238, August 1983.

[S84] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems*, Vol. 2(2), pp. 145-154, May 1984.

[SSKBXI01] D.T. Stott, N.A. Speirs, Z. Kalbarczyk, S. Bagchi, J. Xu, and R.K. Iyer, "Comparing Fail Silence Provided by Duplication versus Internal Error Detection for DHCP Server", *IEEE 15th Int. Symposium on Parallel and Distributed Processing*, San Francisco, April 2001.

[VC02] P. Verissimo and A. Casimiro, "The Timely Computing Base Model and Architecture", *IEEE Transaction on Computing Systems*, 51(8), pp. 916-930, 2002.

[V03] P. Verissimo, "Uncertainty and predictability: Can they be reconciled?", *Future Directions in Distributed Computing,* Springer-Verlag LNCS 2584, 2003.

Appendix A. Implementation of Fail Signal Processes

The construction of fail-signal processes is based upon our earlier work on the construction of *fail-silence* processes [BESST96, BLS98]. The key idea in the construction of a fail-silent process is similar to that of fail-signal processes (mentioned in section 2.1) except that no fail-signals are emitted. A fail-silent process (or object) is made up of a self-checking process (or object) pair. The pair receives the same set of requests in the same order, compute the requests, and then compare each other's results. If the results differ, the replicas stop functioning and refrain from propagating any output to the environment. The fail-silent process has been implemented in both C++ and Java. It was subject to fault injection experiments [SSKXBI01] and no breaches of the fail-silent property were observed. For this paper, the implementation of fail-silent processes was enhanced to include fail-signaling aspect and to run in a CORBA environment so that it can be integrated with NewTOP. The details on the fail-signal implementation are as follows.

Figure A1 shows the internal structures and the inter-workings of the Fail-Signal wrapper Objects (FSOs). An FSO consists of two threads: the Compare process and the replica of *target* process that needs to made into an FS process. *FSO* is termed as the *leader* and *FSO'* as the *follower*. If an input is from another FS process, it is checked for authentic, double signature; this is implemented in the **`receiveNew(m)`** method which, at the leader FSO, places the received input into the local Delivered Message Queue (DMQ) and then sends a copy to the follower by calling the follower's **`receiveDouble(m)`** method.

When the follower receives a message from the leader, it places it into the local DMQ; a copy of the message is also deposited in the Internal Received Message (IRMP) Pool. When the follower receives a valid message via **`receiveNew()`** method, it performs a different task to that executed by the leader. As it gets the message, it checks if the message is in the IRM Pool and if so, the pair is deleted. Otherwise the follower stores it in the IRM Pool with an associated timeout *t1*. If the message is not received from the leader within *t1*, the follower dispatches the message to the leader by calling the **`receiveDouble()`** of the leader. The message within IRM Pool is given a new timeout *t2*. If this second timeout expires and the message has not been received from the leader, the follower assumes the leader has failed and starts emitting fail-signal to appropriate destinations. In the implementation the *t1* is set to 0, and *t2* is set to 2δ.

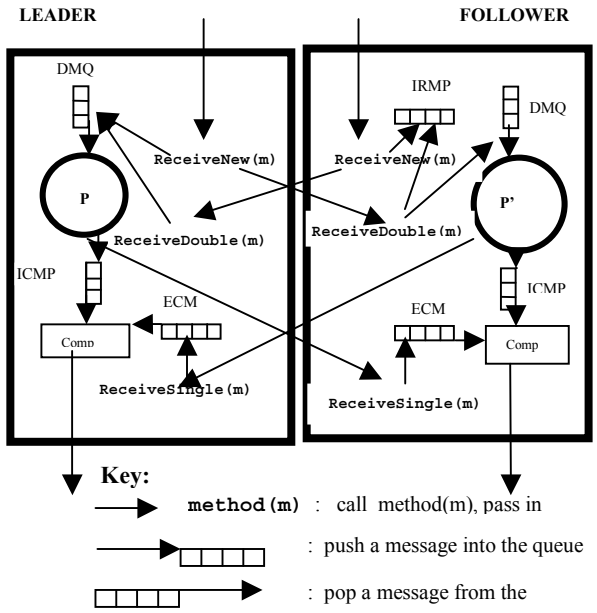

Figure A1: Fail Signaling Wrapper objects

The *target* thread selects a message from DMQ, processes the message, and may produce an output message. A copy of an output message is signed once and transmitted to the other *target* replica by calling the **`receiveSingle(m)`** method of the latter. The unsigned message is stored in the Internal Candidate Message Pool (ICMP), setting a timeout. The timeout duration was computed as described in section 2.2 with $\kappa = \sigma = 2$. When a single signed message is received, it is placed in the External Candidate Message Pool (ECMP). The Compare thread compares relevant messages in ICMP and ECMP. If the comparison indicates that both messages contain identical result, then the comparison is deemed successful, the message from the ECMP is signed again, and the doubly signed message is sent to the destination(s). If the comparison fails or if an ICMP entry times-out, the Compare thread starts emitting fail-signals to appropriate destinations.

Observe that the simple, asymmetric, leader-follower arrangement guarantees message ordering when the leader is correct. If the leader is faulty, any out-of-order processing will manifest as a failure in the output comparison, causing the follower FSO' to start emitting fail-signals.