Crash and Authenticated Byzantine Fault Tolerance: A Fail Signaling Approach

Dimane Mpoeleng

Computer Science Department
University of Botswana

Abstract

Group communication middlewaresystems are particularly useful in supporting replication and thus in building dependable services. Many such 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 minor 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 deterministic total ordering without liveness requirements at the service provisioning level. We demonstrate our claims of seamless deployment by porting a crash-tolerant CORBA group communication service. We additionally measure the performance of the resulting system and examine the trade-offs between performance and the rigor with which the fail-signal abstraction can be built.

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 within some finite time but cannot guarantee an upper bound on 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 fail even in a benign manner (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). Group-splitting obviously reduces the fault-tolerance potentials of a group; furthermore, 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].

The non-partitionable approach (e.g., [FGS98]) does not allow logical partitioning to occur. The total order protocols used can be deterministic or probabilistic in nature. Termination of a deterministic 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. Such system-level requirements for termination are usually called the liveness requirements. We refer the reader to [CT96] which presents a family of gradually-weakening requirements that are termed as failure detectors. For example, a principal requirement of the weakest failure detector (commonly denoted as $\omega$) is that there must be an operative process and a timing instance during an execution such that the former is not suspected between the latter and the instance when the execution terminates. Total-ordering latency is influenced by how early the properties of failure detectors used are realized during the protocol execution which, in turn, depends usually on the choice of timeout values chosen to suspect crashes. For example, in a $\omega$-based system, when timeout chosen is small compared to actual message delays, it postpones the realization of $\omega$ 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.

Termination of non-deterministic or randomized protocols, on the other hand, requires that the random choices made independently by processes converge; this requirement is guaranteed in probabilistic terms to be a certainty with the passage of time [EMR01]. The more different are the values chosen by distinct processes, the longer it will take for the convergence to occur. Thus, the asynchronous, crash-tolerant protocols – be of partitionable or non-partitionable types - make it hard to provide meaningful performance guarantees to applications they support. The difficulties are exacerbated when non-benign faults are considered.

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, which would naturally be influenced by inter-node communication delays, should ideally be free of other factors which could affect it in an arbitrary manner (responsiveness). Given that (ii) is achieved, it becomes relatively easier to predict performance and offer performance guarantees. We note that the focus here is on enhancing robustness and on performance evaluation of an enhanced system, not on performance prediction. We refer the interested readers to [V03] for the difficulties involved in reconciling performance predictability and the uncertainty embodied in the FLP impossibility result, which essentially arise due to the combination of asynchronous communication and the need to tolerate failures.

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., derive a Byzantine-tolerant protocol from a crash-tolerant one [BHR00] and the derivations focus on structural modifications necessary on the latter. Implementing these modifications will however require re-structuring the crash-tolerant system and re-programming some of its components. This becomes obvious if we note that all of these Byzantine-tolerant protocols require at least one extra communication round than their crash-tolerant counterparts.

In this paper, we seek an alternative way to deal with the FLP impossibility in our pursuit of enhancing robustness and maintaining responsiveness. This is motivated by two observations. First, the FLP impossibility applies only when failures occur quiescently; this quiescence leads to the inability to deduce whether a process is crashed or is being merely slow. If processes are designed to announce their imminent failures, then the problem of distinguishing between crash and slowness disappears; one could wait as long as it takes to receive a response, since a process is guaranteed to either produce a response or announce its failure and the underlying communication system does not delete a message in transit but delivers it within some finite time.

Secondly, it is not impossible to construct processes that signal their own failures. The pioneering work on this problem can be seen in [SS83, S84] and the resulting processes adhere to the well-known fail-stop abstraction. In this paper, we build a middleware process as a pair of self-checking and Byzantine-prone
processes on distinct nodes, and the construction leads to fail-signal abstraction. The failure characteristics of a fail-signal process are stated below.

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

- will output its fail-signal whenever it cannot produce a correct response
- can output its fail-signal as well as a correct response, and
- can output its fail-signal at arbitrary instances.

Note that a subtle but benign form of 2-facing behavior is possible. When a faulty process sends fail-signal to some destinations and correct responses to others, some will conclude on the failure of the source process and some others will not. This is similar to a crash-prone process crashing while multicasting a message, leaving some destinations to receive the message and others to either await a message or suspect a crash. By comparison, a fail-stop process [SS83] 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 suffices for constructing a fail-signal process. Note that since a signaling process is necessarily faulty, another process that receives a fail-signal can correctly regard the source to be faulty; i.e., failure detection does not involve choosing appropriate timeouts for suspicions. Thus, with the fail-signal 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 failure detectors (or similar requirements [CL99]).

The cost aspect of our approach can be estimated as follows. Masking off 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 fail-signal middleware processes and each such process itself is a self-checking pair running on distinct nodes, 4f+2 nodes are needed i.e., (f+1) nodes more than the known optimal requirement for a Byzantine-tolerant middlewaresystem. We believe that this cost will be small, given the falling hardware price.

Demonstrating the benefits which the additional hardware is to offer is the major contribution of the paper. The benefits observed are two-fold: simplified construction of a Byzantine tolerant middleware by upgrading a crash-tolerant middleware through software re-use, and the performance degradation due to this upgrading of fault-tolerant capabilities. The latter aspect is measured for three types of authenticated Byzantine failures: a faulty process cannot undetectably alter the contents of a message signed by a correct process, if the message signature has been generated by

- the RSA scheme using a 1024-bit key,
- the RSA scheme using a 512-bit key, and
- the HMAC-MD5 Hash algorithm [BCK96].

We observe that as the signature scheme used becomes less rigorous, the overhead for signature generation and authentication falls and the performance degradation decreases noticeably. As a matter of interest, we evaluated the degradation using no signature mechanisms at all. (The resulting system assumes that faulty processes fail by omitting to respond or by outputting messages which are self-evidently incorrect [GPSS04], and that such outputs are ignored at the destinations.) Interestingly, the resulting degradation does not differ much from that observed when the hash algorithm is used.

This paper is an improved version of [EMS03] in which we considered only the RSA scheme with 512-bit keys. It is organized as follows. Next section outlines the overall approach and describes the construction of fail-signal processes, FS processes for short, together with the assumptions involved. Section 3 describes the
existing NewTOP software and extends it to the FS-NewTOP system. Section 4 presents the experimental set-up and measures the performance of both NewTOP and FS-NewTOP systems. Section 5 concludes the paper.

2. The Replication Context and the FS Approach

We will denote a crash prone node as $C$-node and an authenticated-Byzantine fault prone one as $AB$-node. Figure 0(a) depicts the conventional placement of Group Communication (GC) middleware in replication for crash tolerance: between applications and communication subsystem. The latter is an asynchronous one – offering no bound on communication delays. AGC system typically offers a variety of services such as reliable multicast, uniform reliable multicast, total-order multicast, virtually synchronous group view updates, etc. Let $A_1$ and $A_2$ in Figure 0(a) are two distinct applications that are actively replicated [S93].

Let us suppose that the active replication system needs to tolerate at most $f$ node crashes. Replicas of application processes, such as $A_1$ and $A_2$, need to be hosted by at least $(f+1)$ $C$-nodes. However, the replicated system must have at least $(2f+1)$ $C$-nodes in order that concurrent requests be submitted to replicas in the same order. An optimization may mean that $f$ of these nodes need not act as replicas for a given application, but take part only in the ordering of requests for that application. (This in turn means that a given $C$-node can host replicas of only a subset of applications.)

If an upgrade to Byzantine fault tolerance is desired (for the same $f$), a pair of $AB$-nodes of Figure 0(b) will replace each $C$-node of Figure 0(a). For clarity, let us distinguish the two $AB$-nodes as the left and the right nodes. $GC'$ is a replica $GC$, with the dash being used only to indicating that its host is the right node. Fail-signal (FS) wrappers ensure that $\{GC, GC'\}$ have fail-signal failure modes. (The details are explained in the next subsection.) Hence, the pair $\{GC, GC'\}$ wrapped within an FS wrapper at distinct nodes is denoted as the FS GC.
Typically, one half of the application processes can be hosted by the left AB-node and the other half by the right node. When $A_i$ invokes a GC service, the left FS wrapper transparently traps and passes the invocation to the right wrapper, and the invocation is thus transparently made both on GC and GC'. Thus, each $A_i$ – be on the left or the right node – invokes the FS GC system for a GC service. Similarly, a given GC service message is delivered both by GC and GC' to respective wrappers, and a wrapper forwards the delivered message if the destination $A_i$ is locally hosted or discards it otherwise. (Further details are provided in Section XX.)

Recall that FS GC either provides the required GC service correctly or fails in the fail-signal manner; thus inputs to the replicas of $A_i$ are totally ordered if the respective host AB-node pairs are correct. Given that a given $A_i$ replica can produce erroneous results, we require $(2f+1)$ AB-node pairs so that the erroneous results can be masked through a majority vote. Thus, active replication in the presence of at most $f$ AB faults requires $(2f+1)$ nodes in total. (See also the caveat below.)

We remark here that a crash-tolerant system can also be built using the primary-backup or passive replication approach [BMST93]. In this approach, only one replica (the primary) performs the actual computation and updates the process states of remaining replicas (back-ups); when a primary failure is detected, one of the correct replicas take-over the role of the primary (fail-over) such that there is at most one primary at any given time. Tolerance to at most $f$ node crashes obviously requires at least $f$ back-ups; however, when nodes are connected by an asynchronous communication subsystem, a consensus protocol is required to agree on suspected primary crash and to ensure that the fail-over is done correctly. It is well-known that consensus and total ordering are equivalent [CT96], and therefore at least $(2f+1)$ C-nodes are needed. Of course, the optimization mentioned for active replication is equally applicable here as well.

Upgrading a primary-backup crash-tolerant system to a Byzantine tolerant one requires two major and related difficulties to be overcome. First, the types of node faults that can be tolerated are restricted to crash, or omission at the most where a fault replica may omit to receive incoming messages or send its output messages; specifically, faults that can cause a replica to respond too late or with erroneous results cannot be tolerated. This restriction is inherent to this approach as backup replicas do not do any processing but simply adopt the state updates indicated by the primary.

Secondly, when the primary can fail by outputting erroneous state updates, detecting a primary failure involves carrying out the computation itself on a separate, correct node and checking the outcomes of the two executions. The problem is that the application processes of a primary-backup crash-tolerant system need not be a deterministic state machine: execution of an operational step in a given state and with a given set of arguments can produce one of many possible distinct results, all of which are equally likely in a given execution. That is, when the outcomes of redundant executions are different, it is impossible to attribute the difference unambiguously either to non-determinism or to a fault. We refer the reader to [Powell Delta 4, Priya SRDS04] for the measures that can be taken to identify the sources of non-determinism in the application code and make the latter behave like a deterministic state machine. Given that the application processes are, or can be made to behave like, deterministic state machines, they can be treated as $A_i$ of figure 0(a) and an upgrade to Byzantine fault tolerance can be obtained as suggested earlier.

### 2.1 From GC to Fail-Signal (FS) GC

**Assumptions and Principles**

To transform a middleware process $p$ into fail-signal $p$, or an FS $p$ for short, $p$ 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 $\{p,$
We make the following assumptions.

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 $\delta$ (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 $\max\{\Delta t, \Delta t'\} \leq \kappa \cdot \min\{\Delta t, \Delta t'\}$, for some known, positive number $\kappa$ (A3). Similarly, suppose that both nodes are non-faulty and $p$ and $p'$ schedule a $\text{send\_result()}$ 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 $\max\{\Delta s, \Delta s'\} \leq \sigma \cdot \min\{\Delta s, \Delta s'\}$, for some known, positive number $\sigma$ (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 $\{\text{Order}, \text{Order}'\}$ and $\{\text{Compare}, \text{Compare}'\}$ which are threads (like $p$ or $p'$) within a Fail-signal wrapper Object pair $\{\text{FSO}, \text{FSO}'\}$ (see figure 1). A message destined for FS 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.

We remark that the probability that the fail-signal abstraction is maintained depends critically on the signature mechanism that is used. The more secure (and hence time consuming) the signature mechanism used, the more likely it is that the fail-signal abstraction will be upheld. For non-intrusive failure, a relatively lightweight signature mechanism may be appropriate.

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 [SSKXXBI01]. We have modified our fail-silence implementation to augment it with the fail-signaling feature and to make the system CORBA compliant.

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 has been subject to fault injection experiments [SSKXXBI01] and no breaches of the fail-silent property were observed. The implementation of fail-silent processes has now been enhanced to include a 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 2 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 (IRM) 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.
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 $\delta$.

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

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 3. 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 Aᵢ multicasts a message to the group, the message is marshaled into a generic CORBA type \textit{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 \textit{any}) and delivers it to the client application Aᵢ.

If a host node of, say, Aᵢ in figure 3 develops Byzantine faults, the faults can manifest at two levels. First, at the \textit{application level}, the message which Aᵢ multicasts to the group may contain erroneous information. To tolerate this failure, A₂ and A₃ must be replicas of Aᵢ; given that the latter are correct, the failure of Aᵢ can be masked through a majority vote. Secondly, the fault may manifest at the \textit{middleware level}. The NSO associated with Aᵢ, when hosted in the same node as Aᵢ, may corrupt, probably undetectably, Aᵢ’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 4 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 NewTOPGC, either from the Invocation layer or from a remote NewTOP GC, is intercepted on the fly and is submitted to both \textit{GC} and \textit{GC’} in an identical order with the \textit{FSO} acting as the leader. Similarly, a double-signed response returned by \textit{FSO} and \textit{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, \textit{GC} and \textit{GC’} run as deterministic state machines regardless of other software (e.g. CORBA) running on the host nodes.
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. Further, 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-signal 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 $\Diamond w$ based (crash-tolerant) protocol, by the fail-signaling properties of FS-GCs, the requirements of $\Diamond w$ 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 $\Diamond w$ based, crash-tolerant total-order protocol for a (mixed) system of $f$ FS processes and $(f+1)$ Byzantine-prone processes.

Figure 5: Deployment of the components of FS-NewTOP for a 3-Member Group.

Figure 5 shows the deployment of FS-NewTOP for a 3-member application group $\{A_1, A_2, A_3\}$. For a given $i$, $1 \leq i \leq 3$, $FSO_1$ and $FSO'$ are placed in nodes connected by a synchronous LAN; they are also connected to every $\{FSO, FSO'\}$, $i \neq j$, 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 $FSO'$ is faulty, $A_1$ cannot effectively communicate with the rest of the group due to $FSO$, having stopped the middleware operations or fail-signals being emitted arbitrarily (by $FSO'$). If the node of an $FSO$ is faulty, the following failure mode is also possible: $A_1$ 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 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 (except in section 4.3) 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 6). 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. The machines are networked together in a stand alone configuration ensuring that no external traffic can influence the results. The nodes were running Linux 2.4 and Java 1.4.2. We evaluated the overhead for groups of up to 15 members, using 15 nodes and leaving one node for the collection of performance statistics. We have measured the performance for five different system configurations. The five systems are: NewTOP without fail signal infrastructure, FS-NewTOP without message signatures FS-NewTOP with messages signed with 512 bit RSA key size, FS-NewTOP with messages signed with 1024 bit RSA key size and FS-NewTOP with messages signed by the HMAC-MD5 hash algorithm [BCK96]. The HMAC-MD5 algorithm uses the MD5 cryptographic hash function to provide an authentication code for each message. This hash function is (as we shall see) less time consuming to compute than RSA based message signatures. These five configurations allow us to measure the overhead of the fail-signal architecture and assess it’s dependence on the signature mechanism that is used.

4.1. Throughput

The first experiment measures the throughput of the five system configurations under maximum message load. The results were obtained by fixing the size of messages to 5 bytes, and then each member sending as many messages as possible to the group. We obtain a count of messages per second. This scenario is carried out on each of the five systems: NewTOP, FS-NewTOP without digital signatures, FS-NewTOP with
RSA size 512 bits, FS-NewTOP with RSA size 1024 bits and FS-NewTOP with messages signed by the HMAC-MD5 hash algorithm. Figure 7 shows the performance results for all the five group system configurations against the increasing number of group members from 2 up to 15.

Figure 7: Throughput (in msgs/sec) under maximum message load against varying group members

An interesting observation is that all 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 all systems when groups are larger than the size of the thread pool.

As expected, each of the four configurations of FS-NewTOP perform worse than the original NewTOP because all four configurations replicate each of the NewTOP members. The replicas also need the additional fail signal protocol activities such queue management, encrypting and decrypting signatures and handling duplicate messages. FS-NewTOP has an intrinsic throughput overhead of around 20-30% depending upon group size. The signature mechanism provides a further overhead. The HMAC signature mechanism is the best performing signature mechanism. It reduces the throughput of FS-NewTOP by 5-15% depending upon the size of the group. The FS-NewTOP with RSA digital signature perform significantly worse than the HMAC-MD5 hash algorithm. The significant overhead of digital signatures is clearly illustrated in Figure 7. We then measure throughput with the number of group members fixed at 10 but changing message sizes from 0 to 10k bytes. The results are displayed in Figure 8. The experimental environment is the same as before. All five systems significantly lose performance as the message size reaches 4k bytes. As expected the performance of FS-NewTOP systems with large signature processing degrade for large messages.

Figure 8: Throughput under maximum message load against varying message size
4.2. Order Latency

We define order latency to be the time taken (in ms) to symmetrically order messages within a non-faulty group. When messages have been ordered, they are ready to be delivered by NewTOP to the application. The results are obtained by sending messages from each group member at a fixed interval of 250ms. The message size is fixed at 5 bytes whilst the size of group varied between 2 and 15 members. The results are shown in Figure 9 below.

The FS-NewTOP versions perform quite well with a decreasing percentage overhead as group size increases. The overhead of FS-NewTOP without signatures is only 33% compared to NewTOP with 15 members. FS-NewTOP with MAC signatures has an overhead of 45% compared to NewTOP with 15 members and even FS-NewTOP with RSA-1024 signatures has an overhead of just 76% with 15 members. These overheads are not unreasonable and suggest that the fail signal mechanism might well be applicable in high integrity systems.

The same order latency experiment is conducted but with group membership size fixed at 10 and message size varying from 5 bytes to 10k bytes. The latency figures for the different systems are shown in Figure 10.

![Figure 9: Symmetrical order latency against varying group members](image)

![Figure 10: Symmetrical order latency against varying message size.](image)
FS-NewTOP shows an increasing latency difference to NewTOP. For small groups the overhead is between 1 and 3 seconds depending upon the signature mechanism used. 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. As the message size increases, the signature processing load on nodes increases, resulting in an increasing difference.

### 4.3. Cost of Group Membership when One Member Fails

This experiment measures the time in milliseconds, which is taken for the remaining non-faulty group members to reach an agreement of the new consistent membership view at each member, $v_i$, when one member suddenly fails by stopping. Messages were sent from each member, and one group member was crashed during this process. The time is then recorded from the moment one member crashed until all members agreed upon new membership. This process was repeated 10 times and averages were collected. We start with a group size of three members (not two) so that we are left with a minimum of two members when one member crashes.

The results are shown in Figure 11. The results in Figure 11 indicate a performance advantage for FS-NewTOP over the traditional NewTOP. Note that because a signaled failure from an FS-NewTOP is a guaranteed failure, the receiving members do not have to negotiate with other members in order to establish if indeed the member $M_f$ has actually failed. When a non-faulty member receives a fail signal message, it immediately updates its view without requiring group view consensus broadcasts to the other members. That is why in this experiment the pure NewTOP performs worse in group membership protocol than all four FS-NewTOP versions.

![Graph](image.png)

**Figure 11: Membership protocol overhead when one member fails, against varying group members**

Finally, we repeat the above experiment but vary the message size and fix the number of group members to 10. This experiment shows that FS-NewTOP with lightweight signatures outperforms NewTOP. However, for large message sizes NewTOP begins to perform better.
Figure 12: Membership protocol overhead when one member.

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. Furthermore changes in group view are agreed more quickly in FS-NewTOP.

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, in Figure 4, can also be made into an FS A, in the same way middleware processes were transformed. This will incur an additional FS overhead at the application level and subject replicas of A to assumptions A3 and A4. Alternatively, when A 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 AI 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.

References


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 (IRM) 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.

![Diagram of Fail Signaling Wrapper objects](image-url)

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.