

# A Middleware Architecture for Intrusion Tolerant Service Replication

Paul Ezhilchelvan

University of Newcastle, Newcastle upon Tyne, United Kingdom.

[paul.ezhilchelvan@ncl.ac.uk](mailto:paul.ezhilchelvan@ncl.ac.uk)

## Abstract

*This paper presents a novel combination of known techniques for building a middleware which can support service replication in a hostile environment where a node can get corrupted and fail arbitrarily and message transfer delays cannot be accurately bounded. Using localised replication and output comparison, fail-arbitrary behaviour is reduced to fail-signal: the middleware process of a corrupted server site fails only by emitting a fail-signal, and eventually fails permanently. With this failure-mode, it is possible to avoid the FLP impossibility result which applies only for crash failures; specifically, the termination of a deterministic asynchronous order protocol can be guaranteed even if network delays fluctuate arbitrarily (due to network intrusions) for an indefinite period. We show how reduction to fail-signal is achieved and present a deterministic, message-ordering protocol. We then argue that several, well-known crash-tolerant order protocols can be re-used with little re-design within the proposed middleware.*

## Introduction

It is an emerging view, reflected also in the workshop call, that the systems built using low-cost commercial components are vulnerable to an intruder exploiting security flaws which remained unnoticed during the system development process. The behaviour of an intruded node can be arbitrary in nature; in particular, it may include the Byzantine failure mode of generating a response with erroneous information in a syntactically valid format. Active replication is known, since the days of SIFT [14], as a promising way of guaranteeing available and high-integrity services despite a bounded number of nodes failing in arbitrary manner. It involves executing a given service process on multiple nodes which have no common point of failures, and ensuring that the correct nodes form a majority and produce identical results for any given service request. The latter in turn ensures

that a majority vote on the replicas' responses returns only the correct response. If it is assumed that an adversary can corrupt, before being noticed, only less than half the number of replicas and that the service processes are implemented by deterministic state machines, then the major requirement for supporting service replication is that the client requests be delivered to service replicas in an identical order. In this paper, we propose architecture for a middleware that meets this requirement.

To order client requests identically, middleware processes may have to interact with each other. In an intrusion-prone environment, the adversary may attempt to block, delay, or corrupt messages on the network without giving rise to a security alert. Therefore, the message transmission delays between correct middleware processes cannot be bounded *a priori* with certainty, unless the network is a trusted private channel. Being general purpose in our design, we will assume that when correct middleware processes send messages to each other, there is no known bound on delays to be experienced, a sent message is eventually received, and a received message is the one that has been sent. In other words, the classical asynchronous delay model is assumed for the interaction between middleware processes of service replicas.

Message ordering is shown in [7] to be equivalent to solving the consensus problem for which a deterministic solution is impossible [6] when a known, fixed bound on delays does not exist and nodes can crash. More precisely, the FLP impossibility result [6] states that a deterministic protocol that can be guaranteed to terminate cannot exist if nodes can crash in the asynchronous delay model. Asynchronous, Byzantine-tolerant protocols published in the literature work around the FLP impossibility in two ways. In the first category,

termination is guaranteed, provided the network delays eventually stop increasing faster than some pre-determined function [3], or eventually stop increasing beyond a dynamically estimated bound [5, 8], for a sufficiently long period. The protocols of the second category, such as [2, 9], do not impose such requirements on eventual network delay variations, but provide only probabilistic termination guarantees. All the protocols cited above assume message signatures and require more than two-third of middleware processes to remain correct which is the known lower bound.

We pursue an approach in which a middleware process, say  $p_i$ , is built with internal redundancy so that it can fail only in a specific manner, called the *fail-signal*, defined below:

- *Signal on failure*:  $p_i$  fails only by emitting a distinct exceptional response *fail-signal*, and
- *Eventual permanent failure*: There exists an unknown time after which any response of faulty  $p_i$  is only *fail-signal*.

The visible behaviour of  $p_i$  falls into three epochs of unknown duration: *correct* period in which only correct responses are produced, a transient *failing* period in which both correct responses and *fail-signal* may be produced and the *failed* period in which only *fail-signal* is generated. A *fail-signal*  $p_i$  thus differs from a *fail-stop* process [12] in two ways: the latter has no *failing* period and retains the (correct) state in which the *fail-signal* was first emitted; its construction also requires more redundant nodes than what is needed for *fail-signal*.

Constructing middleware processes to be *fail-signal* is motivated by the following factors. (i) If a process that receives a *fail-signal* from  $p_i$  assumes  $p_i$  to have crashed and ignores any subsequent responses from  $p_i$ , then its assumption eventually becomes true by the eventual permanent failure property. Thus, the *fail-signal* failure model is stronger than the crash model in the sense that failures can be reliably detected without having to rely on network delay variations to conform to assumptions made in dynamic delay estimations. This allows deterministic order protocols to terminate even if network delays fluctuate arbitrarily. (ii) That correct processes can reliably detect the failed ones can be used to facilitate on-

line replacement of failed ones by spare processes. An arrangement for dynamic reconfiguration of the replica system naturally increases system's resilience to further intrusions.

The paper is structured as follows. Next section describes the system model and assumptions. Section 3 describes the middleware architecture for constructing *fail-signal* processes. Section 4 describes the asynchronous order protocol executed by middleware processes for the ordered delivery of client requests. This protocol is used to illustrate that the failure detector  $\diamond_w$  of [4] – the weakest necessary to deterministically circumvent the FLP result - can be implemented using *fail-signal* processes without relying on timeout estimates. Section 5 compares our approach to the known ones and concludes the paper.

## 2. System Model and Assumptions

The system consists of  $n$ ,  $n > 2$ , *server sites* interconnected by an asynchronous network such as the Internet. Message communication over the network is reliable in the sense that every sent message is eventually delivered exactly once to its destination(s) and a delivered message has been sent by some server. Communication delays are finite but cannot be accurately bounded at any time. Specifically, we will make no assumption on how delays could change over time or whether they would remain stable during an interval of specified duration.

Each service provided by the system is replicated over all  $n$  sites. We assume, for simplicity, that a given client request requires the provision of only one service. Figure 1 shows a system of 3 sites ( $S_1$ ,  $S_2$ , and  $S_3$ ) offering four distinct services implemented by deterministic processes  $s_1$  to  $s_4$ . The order process  $p_i$  of site  $S_i$  delivers client requests to appropriate  $s_i$  in an identical order. In our service model, a subset of the service state maintained by  $s_i$  can be modified by an arbitrary number of authorised clients. (In some applications, e.g. [15], a given subset of the service state can be modified only by a given client; in such cases, the ordering requirement is relatively easy to meet.)

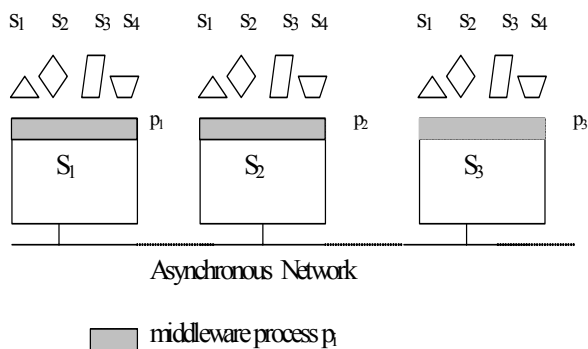


Figure 1. Replication of four services over  $n=3$  sites.

Each server site is managed by a trusted administrator who takes all known measures to prevent an attacker from penetrating the site boundary. Despite the preventive measures taken by the site administrator, a node in a given site can become corrupt due to an attack and fail arbitrarily. We assume that at most  $f, f < n/2$ , sites will have a corrupted or a faulty node within them. Thus, service process replicas in at least  $(n-f)$  sites will generate correct and identical responses to client requests.

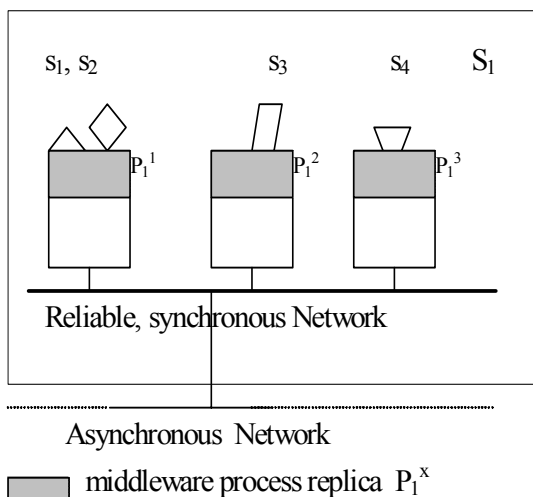


Figure 2. Internal structure of site  $S_1$ .

Each site  $S_k, 1 \leq k \leq n$ , is made up of  $b_k$  nodes,  $b_k \geq 1$ , of which maximum of  $\{1, (b_k-1)\}$  nodes can simultaneously become faulty or get controlled by an attacker. If  $b_i > 1$ , the nodes of site  $S_i$  are interconnected by a secure, reliable and

synchronous LAN or inter-LAN; the order process  $p_i$  is replicated over all  $b_i$  nodes so that it suffers only fail-signal failures. The following assumption renders a synchronous environment which is necessary to build  $p_i$  to be fail-signal:

The relative processing speeds of, and the message transfer delays between, correct replicas of  $p_i$  are bounded by known constants.

A replica of  $p_i$  is termed correct when its host node is correct. Figure 2 shows site  $S_1$  with three ( $b_1=3$ ) nodes, each hosting a distinct service replica. The middleware process  $p_1$  of  $S_1$  is thus an abstract entity (group) made up of three replicated processes:  $P_1^x, 1 \leq x \leq 3$ . Replica management will ensure that the failure mode of  $p_1$  is fail-signal.

We assume that all nodes in each site are correct when the system initially starts up; further, replication degree  $b_k$  of each site  $S_k$  and the public key [11] of each node is known to all nodes in the system.

### 3. Middleware Architecture

Figure 3 shows the three middleware components within a node  $x, 1 \leq x \leq b_i$ , of site  $S_i$ : input/output (*io*) process, the *order* process ( $P_i^x$ ), and the *fail-signal layer* which contains a process pair  $\{R_i^x, M_i^x\}$ . The fail-signal layer hides the details of replication (if any) enabling  $P_i^x$  to regard itself as ‘the’  $p_i$  and to regard  $p_j$  of another  $S_j$  as a single (non-replicated) process. It also ensures that the group  $\{P_i^y, 1 \leq y \leq b_i\}$  exhibits only fail-signal failure modes if  $b_i > 1$  and no more than  $(b_i-1)$  nodes fail. The architecture presented here is a simplified version of the fail-stop architecture in [12] and an enhancement over the fail-silent architecture in [1].

The *io* forwards the client requests (type 1 in fig. 3) to  $R_i^x$ . Every  $P_i^y, 1 \leq y \leq b_i$ , implements a deterministic, asynchronous order protocol (described in Section 4) and receives identical messages in identical order from its respective  $R_i^y$ .  $\{R_i^y, 1 \leq y \leq b_i\}$  execute a non-fault-tolerant order protocol over the synchronous channel. The worst case overhead for synchronous ordering (when  $b_i > 1$ ) is: 1 broadcast (by the sequencer) and 1 unicast (from a non-sequencer to the sequencer).

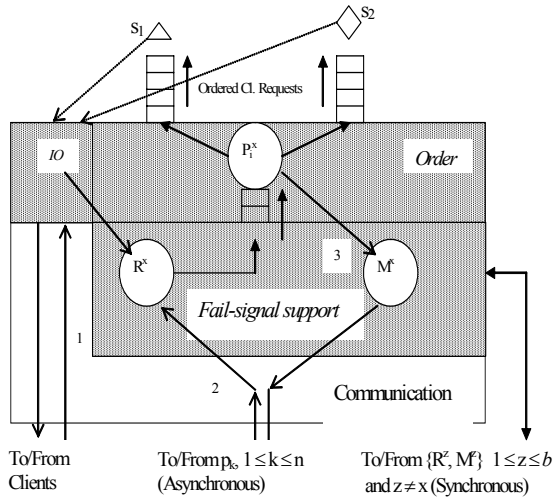


Figure 3. The middleware structure.

Process  $R_i^x$  accepts a message (type 2 in fig. 3) from  $p_j$  only if the received message has authentic signatures of all  $b_j$  nodes of site  $S_j$ . If the accepted message is a *fail-signal<sub>j</sub>* from  $p_j$ , it diffuses *fail-signal<sub>j</sub>* once to all  $p_k$ ,  $1 \leq k \leq n$ ; if  $j = i$ , it also instructs  $M_i^x$  to stop functioning.

For every message  $m$  (type 3) produced by  $P_i^x$  for any  $p_j$ ,  $M_i^x$  is responsible for obtaining an authentic signature for  $m$  from each  $M_i^y$ ,  $1 \leq y \leq b_i$  and for transmitting  $m$  to  $p_j$  together with  $b_i$  signatures. Note that a remote  $M_i^y$  does not return its signature for the  $m$  sent by  $M_i^x$  until a matching  $m$  is locally produced by  $P_i^y$ . The output authentication overhead over the synchronous channel (when  $b_i > 1$ ) is:  $b_i$  broadcasts.

If  $M_i^x$  does not succeed in obtaining signatures from all nodes of site  $S_i$  within certain timeout, it emits *fail-signal<sub>i</sub>* to all  $p_k$ ,  $1 \leq k \leq n$  and then stops functioning. We assume that, at the time of site initialization when all nodes are correct, the M-processes exchange a signed *fail-signal<sub>i</sub>* with each other. Thus, each  $M_i^y$  is in ready possession of valid *fail-signal<sub>i</sub>*. Note that this arrangement empowers an M-process of a compromised or faulty node to transmit *fail-signal<sub>i</sub>* at any time and to any destination of its choosing, while at the same time behaving correctly towards its counterparts in the site. This gives rise to the transient failing period when  $p_i$  can output both *fail-signal<sub>i</sub>* and correct responses. Since *fail-signal<sub>i</sub>* is diffused back to the originating site, eventually correct M-processes of

that site stop functioning. Thus, the *eventual permanent failure* property is maintained.

When  $P_i^x$  orders a client request, it puts the ordered request in the appropriate queue if the request is destined for a locally-hosted service replica (such as  $s_1$  or  $s_2$  for  $P_i^1$  in figure 2); otherwise, it discards the request.

## 4. An Asynchronous Order Protocol

We present the protocol for a set  $\pi$  of  $n$ ,  $n = 2f + 1$ , processes  $\{p_1, p_2, \dots, p_n\}$  of which at most  $f$  can fail. The subset FS,  $\{p_1, p_2, \dots, p_f\}$ , is a set of internally replicated processes (shown as shaded circles in figure 4) which fail only in the fail-signal manner, while the rest, the normal ones,  $NL = \{p_{f+1}, p_{f+2}, \dots, p_n\}$  are not replicated and can fail arbitrarily. Processes are uniquely ranked, say as  $p_1, p_2, \dots, p_n$ , with a fail-silent process preceding any normal process in the ranking. The ranking is known to all processes.

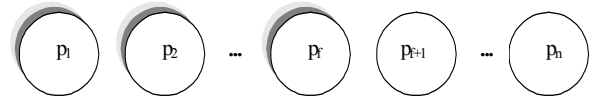


Figure 4. System of  $f$  fail-signal and  $(f+1)$  normal order processes.

The protocol is a coordinator- and quorum-based one. At the start,  $p_1$  – the highest ranked – acts as the coordinator. A process  $p_i$ ,  $2 \leq i \leq f+1$ , can become a coordinator only after it has known that all  $p_k$ ,  $1 \leq k \leq (i-1)$ , have fail-signaled. When this is enforced during any coordinator change,  $p_{f+1}$  – the only normal process in the coordinator candidate list – becomes the coordinator only after  $f$  site failures have already happened; so  $p_{f+1}$  will be correct and not fail. So, in our protocol, the coordinator is entrusted to order client requests on behalf of the rest of the processes.

A quorum  $Q$  constructed by a correct  $p_i$  at time  $t$  must (i) have  $n-f$  processes in total, and (ii) admit only  $\phi+1$  normal processes where  $\phi$  is the number of fail-signal processes that are known to  $p_i$  as failed at  $t$ . That is,  $Q$  must contain all fail-signal processes which, according to  $p_i$ , have not signaled until  $t$ .

## 4.1. The Protocol

Part I: Each  $p_j$  executes the following four steps:

1. Forward any new, unordered client request to the coordinator;  
     // if  $p_j$  is the coordinator, it forwards the request to itself //
2. receive ordering information  $ord(\mu)$  for some client request  $\mu$ ;
3. after sending an *ack* for all preceding  $ord(\mu')$ , send an *ack* for  $ord(\mu)$  by multicasting ( $ack_j, ord(\mu)$ );
4. after receiving *ack* for  $ord(\mu)$  from quorum  $Q_j$ , deliver  $\mu$  as per  $ord(\mu)$ ;

Part II: The coordinator executes this step concurrently:

1. for every new, unordered client request  $\mu$  received:  $\{ord(\mu) := last\#\#\#;$   
     multicast( $ord(\mu)$ );}

Part III: When  $p_i$ ,  $2 \leq i \leq f+1$ , has received *fail-signal<sub>k</sub>* for all  $p_k$ ,  $1 \leq k \leq (i-1)$ :

1. multicast  $\{ fail-signal_k: 1 \leq k \leq (i-1) \}$  to claim coordinatorship; // any process that receives the above multicast, will temporarily stop acknowledging  $ord(\mu)$  issued by the current coordinator and send to  $p_i$  *last-ack* = the last  $ord(\mu)$  it acked //
2. receive *last\_ack* from quorum  $Q_i$ ;
3.  $last\# = \max\{last\_ack \text{ from } Q_i\}$ ;
4. multicast the end of previous coordinator for  $last\#$ ; // processes resume acknowledging  $ord(\mu)$  of the old coordinator upto  $last\#$ ; //
5. become the new coordinator;

## 4.2. Correctness Hints

We need to show that when  $p_i$  becomes the coordinator, the  $last\#$  it computes by receiving *last-acks* from quorum  $Q_i$  of processes, cannot be smaller than  $ord(\mu)$  of any  $\mu$  which a correct  $p_j$  has delivered after receiving *acks* for  $ord(\mu)$  from a quorum  $Q_j$  of processes. If  $Q_i \cap Q_j$  has a fail-signal process, then that process has responded to both  $p_i$  and  $p_j$ ; therefore, it must have faithfully reported to  $p_i$  about it having acknowledged  $ord(\mu)$ , so we are done.

Say  $Q_i \cap Q_j$  contains no fail-signal process. We show that  $Q_i \cap Q_j$  must contain at least one correct normal process, by arguing: (i) all normal processes are correct, and (ii)  $Q_i$  and  $Q_j$  have at least one common normal process.

Let  $Q_i = FS_i \cup NL_i$ , and  $Q_j = FS_j \cup NL_j$ , where  $FS_i$  and  $NL_i$  are disjoint subsets of  $Q_i$ ,  $1 = i$  or  $j$ , and subsets of FS and NL respectively. Recall that a fail-signal process is not counted in a quorum only if it is known to have fail-signaled. So,  $FS_i \cap FS_j$  is empty implies that all fail-signal processes,  $f$  in total, have failed; further,  $|FS - FS_i| + |FS - FS_j| \geq |FS| = f$ . Secondly,  $|NL_i| - 1 = |FS - FS_i|$ , and  $|NL_j| - 1 = |FS - FS_j|$ .  $|NL_i| + |NL_j| - 2 = |FS - FS_i| + |FS - FS_j| \geq |FS| = f$ . So,  $|NL_i| + |NL_j| > f+1 = |NL|$ .

## 4.3. $\diamond w$ based Order Protocols

We show that the context of the protocol presented above implements a useful version of  $\diamond w$  – the weakest requirement for solving consensus deterministically. The weakest failure detector is characterized [4] by *accuracy* and *completeness* requirements. *Eventual accuracy*: there is a time after which some correct process is never suspected by any correct process, which can be trivially met in a system where a majority never fails if no process suspects any process. So, we have *non-triviality* or *completeness*: any crashed process is permanently suspected by some correct process.

Let us consider a system depicted in figure 4 where at most  $f$  processes can fail and processes are uniquely ordered as in the protocol: a fail-signal process is ordered before a normal one, the rank of the first process is 1, the rank of the second process is 2, and so on. To see that the *eventual accuracy* is met in such a system, let us name the process that is eventually never to be suspected by any correct process, as the *Queen* and denote it as  $Qn$ . Let every process  $p_i$  initially regard the first process as its *Queen*  $Qn_i$ , and reset  $Qn_i$  to the next process only if it receives the *fail-signal* of the current  $Qn_i$ . Let  $rank(Qn_i)$  be the rank of  $Qn_i$ , and  $p_i$  and  $p_j$  be any two correct processes. We need to show that eventually:  $Qn_i = Qn_j$  and  $Qn_i$  is correct i.e., never fails. We can show this by arguing that (i) neither  $rank(Qn_i)$  nor  $rank(Qn_j)$  can exceed  $(f+1)$ , and (ii)  $rank(Qn_i)$  and  $rank(Qn_j)$  cannot differ permanently,

based on the assumption of at most  $f$  failures, on the properties of fail-signal and on the way correct processes change their *Queen*.

To use a  $\diamond_w$  crash-tolerant order protocol, it appears to be sufficient that only the notion of quorum (or majority) be defined as in the protocol above.

## 5. Concluding Remarks

The presented architecture takes a two-step approach to the provision of fault- and intrusion-tolerance. It requires a certain number of server sites to be a collection of nodes connected by a reliable and synchronous network. By performing localized replication and output comparison over this synchronous network, arbitrary failures are reduced to fail-signal failure mode at the level where asynchronous message ordering needs to be done; secondly, the fail-signaling feature is exploited to achieve asynchronous ordering without being subject to the FLP impossibility result. To our knowledge, a philosophically similar approach was taken in the DELTA-4 project [10] in which a special network attachment controller (NAC) was constructed (with internal redundancy) primarily to reduce the ‘multi-facing’ aspects of an arbitrarily failed computational node into a uni-faced incorrect behaviour.

The advantages and the disadvantages of our approach can be directly attributed to reducing fail-arbitrary to fail-signal. The benefits are:

(i) Deterministic message ordering over an asynchronous network terminates even if message delays fluctuate arbitrarily over a prolonged period.

(ii) Fail-signaling feature facilitates easy replacement of failed sites with new correct sites, increasing system resilience to further attacks

(iii) Asynchronous message ordering protocols generally run in ‘rounds’ where each round is an attempt to terminate. A round has two phases for crash tolerance, while Byzantine fault tolerance requires at least three phases [3, 5, 8] per round. Though our middleware uses a crash-tolerant order protocol over an asynchronous network, it has the overhead of synchronous message ordering and output authentication within a site. Thus, if asynchronous delays are longer, then our middleware can provide better performance than the

known asynchronous, Byzantine-tolerant ordering protocols [2, 3, 5, 8, 9]. That is, using a high-bandwidth synchronous channel within a (replicated) site can bring ordering delays closer to crash-tolerant ordering latencies.

(iv) For the same number ( $f$ ) of sites that can fail, the number of total nodes required in our approach is the same as the known Byzantine-tolerant protocols if we assume only  $f$  sites have internal replication with  $b=2$ .

The limitations appear to be:

(i) If an intruder succeeds in disrupting the traffic over the synchronous channel of a replicated site, then the site may signal a failure even if no node is corrupted. To reduce this vulnerability, the channel must be a private and secure one.

(ii) The administrators of replicated sites are trusted; in particular, if they collude with intruders, all nodes of a replicated site can become corrupt simultaneously and the fail-signal abstraction cannot be maintained. We are currently working on a protocol that tolerates a bounded number of (unknown) site administrators who can be compromised.

The Fault-tolerant Real-time architecture of [13] has, in its core, a distributed synchronous computing base in which the processing elements of different sites exchange message over a synchronous network. In our case, the synchronous computing base restricted to selected sites only and to supporting replication of ordering process within such sites. This important difference highlights the end objectives which these architectures are proposed for. In [13], the primary aim of the synchronous computing base is to support, and to monitor the progress of, timed computations carried out in a potentially asynchronous distributed system which encapsulates the synchronous core. Our goal however is to tolerate intrusions and hence it is only safe to consider asynchronous delay model in communication between middleware processes of different sites.

## Acknowledgements

Discussions with Santosh Shrivastava and Michel Raynal were very useful.

## References

- [1] F V Brasileiro, P D Ezhilchelvan, S.K. Shrivastava, N A Speirs, and S. Tao “Implementing Fail-Silent nodes for Distributed Systems”, IEEE Transactions on Computers, 45 (11), Nov 1996, pp. 1226-1238.
- [2] C Cachin, K Kursawe, and V Shoup, ‘Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography’ Proc. 19<sup>th</sup> ACM Symposium on Principles of Distributed computing (PODC), pp. 123-32, 2000.
- [3] M Castro and B Liskov, ‘Practical Byzantine Fault Tolerance’, 3<sup>rd</sup> ACM Symposium on Operating Systems Design and Implementation (OSDI), Feb. 99, pp. 173-186.
- [4] T. D. Chandra and S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems”, Journal of the ACM, 43(2), pp. 225 - 267, March 1996.
- [5] A. Doudou, B. Garbinato, and R. Guerraoui, ‘Abstractions for devising Byzantine-resilient state machine replication’, in Proc. 19<sup>th</sup> Symposium on Reliable Distributed Systems (SRDS 2000), pp. 144-152, 2000.
- [6] 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.
- [7] V. Hadzilacos and S. Toueg, “Fault-Tolerant Broadcasts and Related Problems”, in Distributed Systems, (Ed.) S Mullender, Addison-Wesley, 1993.
- [8] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith, “The SecureRing Protocols for Securing Group Communication”, Proceedings of the 31<sup>st</sup> Annual Hawaii International Conference on System Sciences (HICSS), pp. 317-26, Jan. 1998.
- [9] D Malkhi and M Reiter, ‘Byzantine Quorum Systems’, Distributed Computing, 11(4), pp. 203-13, 1998.
- [10] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton, “The Delta-4 Approach to Dependability in Open Distributed Computing Systems,” Digest of Papers, FTCS-18, Tokyo, pp. 246-251, June 1988.
- [11] R. L. Rivest, A. Shamir, and L. Adleman, ‘A method for obtaining digital signatures and public-key cryptosystems’, Communications of the ACM, 21(2), February 1978, pp. 120-26.
- [12] F B Schneider, ‘Byzantine Generals in Action: Implementing Fail-Stop processors’, ACM Transactions on Comp. Systems, 2(2), pp. 145-54, May 1984.
- [13] P Verissimo, A Casimiro, and C Fetzer, ‘The Timely Computing Base: Timely actions in the Presence of Uncertain Timeliness’, 1<sup>st</sup> International conference on Dependable Systems and Networks (DSN00), New York, June 2000.
- [14] J H Wensley et. al., “SIFT: Design and Analysis of a Fault-Tolerant Computer for Air-Traffic Control”, Proceedings of the IEEE, 66(10), pp. 1240-55, Oct. 1978.
- [15] L Zhou, F B Scheneider, R V Renesse, ‘COCA: A Secure Distributed On-Line Certification Authority’, Technical Report, Department of Computer Science, Cornell University, New York.