Vbam - Byzantine Atomic Multicast in LAN Based on Virtualization Technology

Marcelo Ribeiro Xavier Silva, Lau Cheuk Lung, Leandro Quibem Magnabosco, Luciana de Oliveira Rech

[1] Computer Science – Federal University of Santa Catarina – Florianópolis, Brazil,
marcelo.r.x.s@posgrad.ufsc.br lau.lung@ufsc.br leandroqm@gmail.com
luciana.rech@ufsc.br

Abstract. This work presents a BFT Atomic Multicast Protocol (Vbam) whose algorithm manages to implement a reliable consensus service with only *2f + 1* servers using only common technologies, such as virtualization and data sharing abstractions. In order to achieve these goals, we chose to adopt a hybrid model, which means it has different assumptions between components regarding synchrony, and two different local area networks (LANs), a payload LAN and a separated LAN where message ordering happens.

1 Introduction

The difficulty to build distributed systems can be strongly reduced by relying on group communication primitives such as atomic (or totally ordered) multicast [1]. The atomic multicast ensures that messages sent to a set of processes are delivered by all these processes in the same order. Atomic multicast has many important applications such as clock synchronization, CSCW, distributed shared memory, database replication [2], [3], [4] and basis the state machine approach, the main component of many fault-tolerant systems [5], [6], [7], [8].

There exists a considerable amount of literature on total order multicast, and many algorithms, following various approaches, have been proposed to solve this problem. However, in most cases, these algorithms consider system models subject only to crash faults [1], [9]. Very few address also byzantine/arbitrary faults [10], [7]. In general, these relies on a consensus algorithm to agree on messages ordering and need *3f + 1* processes involved in the agreement. There are some works that separate the consensus from the agreement problem creating a consensus service [11], [12].

We present Vbam, a byzantine fault tolerant consensus service for atomic multicast messages. The model and architecture that we propose needs only *2f + 1* servers compounding the consensus service and is based on a hybrid model where there exist variation, from component to component, on the assumptions of synchrony and presence/severity of failures [13], [14], [15]. In this model, there is a

payload LAN in which the clients communicate with the consensus service, and an inviolable LAN where the servers do the ordering. In our proposal we contribute with an improvement to make the consensus service [11] byzantine fault tolerant with low resilience (only $2f + 1$ servers).

2 Related work

The atomic multicast problem has been widely addressed in the past decades [2], [10], [14], [3], [9], [7]. In the majority, the approaches consider that the processes can only crash, i.e., not acting in arbitrary/byzantine manner.

In [10], it is presented Rampart for reliable and atomic multicast in systems with byzantine faults. The algorithm is based on a group membership service, which requires that at least one third of all processes in the current view reach an agreement on the exclusion of some process from the group. The atomic multicast is done by some member of the group, called sequencer, which determines the order for the messages in the current view. In the next view, another sequencer is chosen by some deterministic algorithm. Rampart assumes an asynchronous system model with reliable FIFO channels, and a public key infrastructure known by every process. With the assumption of authenticated communication channels the integrity of messages between two non-Byzantine processes is always guaranteed.

Guerraoui and Schiper proposed in 2001 the generic consensus service [11] for solving agreement problems, including the atomic multicast. This is the base for our proposal. Their model considers a crash-only environment with a consensus service that separates the consensus from the agreement problem to be solved. The system requires a perfect failure detector (that basis the consensus server) and the resilience is a tradeoff with performance, varying depending on the necessity.

In 2006 Correia and Veríssimo showed a transformation from consensus to atomic broadcast [7]. The system model presented assumes a byzantine environment in which up to $f = [(n - 1)/3]$ faults are tolerated. The authors implement a multi-valued consensus protocol on top of a randomized binary consensus and a reliable broadcast protocol. The atomic multicast protocol is designed as successive transformations from the consensus protocol. The atomic multicast is done by the use of a hash vector. Each process of the system proposes values to the consensus vector (that is a vector with the hashes of the messages). The vector consensus protocol decides on a vector X_i with at least $2f + 1$ vectors H from different processes. In the sequence the messages are stored in a set to be atomically delivered in a pre-established order.

In 2010 Pieri et al proposed an extension of the generic consensus service [11] for byzantine environments [12]. The system model proposed has $n_c = 3f_c + 1$ clients and $n_s = 2f_s + 1$ servers, and they make use of virtual machines to provide the generic consensus service. The atomic consensus starts whenever one of the processes called initiator reliably multicast a message mi to the clients set. Upon receiving the message m_i, each client sends a proposal to the generic consensus ser-

vice for m_i. When the servers receive n_c - f_c proposals from clients to the same consensus instance, each server start a consensus protocol. Then, the result of this protocol is relayed to the clients. The importance of this work comes in means of making the generic consensus available for byzantine environments. However, it limits the size of the clients set. Furthermore the system has to deal with the faulty clients, lowering the system resilience. This is acceptable when dealing with generic agreement problems, but for solving the atomic multicast it is expensive.

Table 1 Comparison on evaluated atomic multicast protocols properties

	Rampart[10]	*Guerraoui and Schiper [11]*	*Correia and Veríssimo [7]*	*Pieri et al [12]*	*Vbam*
Resilience for atomic multicast	**3f + 1**	**-**	**3f + 1**	**$3f_c + 1 + 2f_s + 1$**	**2f + 1**
Communication steps	**6**	**5**	**-**	**5**	**4**
Messages exchanged	**6n - 6**	**$3n_c + 2n_c - 3$**	**$18n^2 + 13n + 1 + 16n^2f + 10nf$**	**$2(n_s^2 + 3n_c - n_s - 1)$**	**$3n_s^2 - n_s + n_c + 1$**
Tolerated faults	**Byzantine**	**Crash**	**Byzantine**	**Byzantine**	**Byzant**

3 System model and architecture

The system model is hybrid [14], which is where assumptions of synchrony and presence/severity of failures vary from component to component [13], [14]. In our model, we consider different assumptions for the subsystems running in host, than for the ones running in the guest of the virtual machines that composes the system. In this model, $C = \{c_1, c_2, c_3, \ldots\}$ is a set that contains a finite number of client processes and $S = \{s_1, s_2, s_3, \ldots, s_n\}$ representing a set of servers with n elements that compound the consensus service.

The Figure 1, each consensus service server is hosted by a virtual machine. Each server physical machine has one, and only one, virtual machine as its guest (see Figure 1). The process failure model admits a finite number of clients and up to $f \leq [(n - 1)/2]$ servers incurring failure based on its specifications, presenting byzantine faults [17]: the faulty processes that arbitrary detour from its specification can stop, omit sending or receiving messages, send wrong messages or have any non-specified behavior. However, we assume independence of faults, in other words, the probability of a process having a fault is independent of another fault in any other process. This is possible in practice by the extensive use of diversity (different hardware, operational systems, virtual machines, databases, programming languages, etc.) [18].

The system has two distinct networks, the payload and the controlled networks (see Figure 1), both are local area networks (LANs). The former is asynchronous and is used for application data transfers. There are no assumptions based on time on the payload LAN and it is used for client-server communication. The later, used for server-server communication, is a controlled LAN composed by physical machines, where is implemented a Distributed Shared Register (DSR) [16]. The

consensus service uses the DSR to execute the crucial parts of the consensus protocol. The DSR:

- has a finite and known number of members;
- is assumed to be secure, i.e., resistant to any possible attacks; it can only fail by crashing;
- is capable of executing certain operations with a bounded delay;
- provides only two operations, read and write register, which cannot be possibly affected by malicious faults.

Each physical machine has its own space inside the DSR where its respective virtual machine registers PROPOSE messages, ACCEPT messages or CHANGE messages. All servers can read all register space, no matter who hold the rights to write into it.

We assume that each client-server pair c_i , s_j and each pair of servers s_i , s_j is connected by a reliable channel with two properties: if the sender and the recipient of a message are both correct then (1) the message is eventually received and (2) the message is not modified in the channel [15]. In practice, these properties have to be obtained with retransmissions and using cryptography. Message authentication codes (MACs) are cryptographic checksums that serve our purpose, and only use symmetric cryptography [19], [5]. The processes have to share symmetric keys in order to use MACs. We assume these keys are distributed before the protocol is executed. In practice, this can be solved using key distribution protocols available in the literature [19]. This issue is out of the scope of this paper.

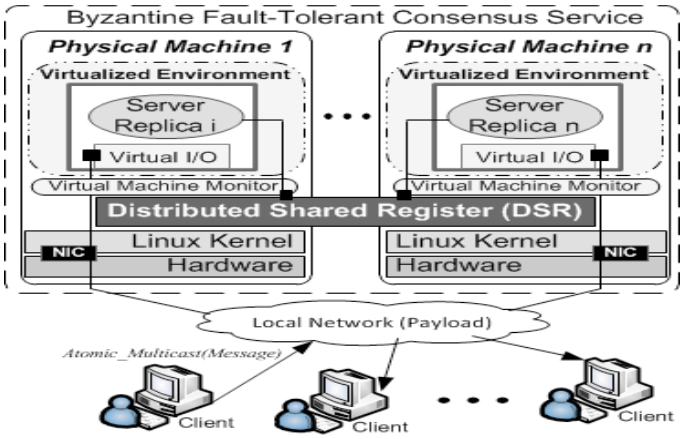


Fig. 1 Vbam Architecture Overview

We assume that only the physical machine can, in fact, connect to the controlled LAN used for the register. That is, the DSR is accessible only by the physical machines that host virtual machines (servers), which are participants of the system, not being possible to reach the DSR directly through accessing the virtual machine. Each process is encapsulated inside its own virtual machine, ensuring

isolation. All client-server communication happens inside a separate LAN (payload) and, on the clients point of view, the virtual machine is transparent, meaning clients can not recognize the physical-virtual architecture. Each machine has only one network interface (NIC), a firewall and/or bridge mode are used in the host to ensure the division of the networks.

We assume that host vulnerabilities cannot be explored by the virtual machine. The virtual machine monitor (VMM) ensures this isolation, meaning the attacker has no way to access the host through the virtual machine. This is a premise in the virtualization technologies, such as VirtualBox, LVM, XEN, VMWare, VirtualPC, etc. Our model assumes that the host system is not accessible externally, which is also granted by the use of bridge mode and/or firewalls on the host system.

Distributed Shared Register (DSR): We created the DSR, an emulated shared memory [16] based on message passing over a controlled LAN and making use of local files. We assume the controlled LAN to be only accessed by components of the DSR. The DSR is implemented in the virtual machine host and we assume that the VMM ensures isolation between the host and the guest.

The DSR performs just two operations: (1) *read()*, that reads the last message written in the DSR; and (2) *write(m)* that writes the message *m* in the DSR. We assume two properties about these operations: (i) liveness, meaning that the operation eventually ends; and (ii) safety, i.e., the read operation always returns the last value written. To ensure this properties, in each server we created a file where the guest has write-only access and another file where it has read-only access and the access is made by a single process [16]. The first file is the server space, and no other server can write on it. The second one is the other servers register, updated by the DSR. The VMM provides the support to make a file created in the host to be accessible to the guest, enforcing the write-only/read-only permissions.

The DSR only accept typed messages, and there is only three types, (i) PROPOSE, (ii) ACCEPT and (iii) CHANGE. The untyped or mistyped messages are ignored. We assume that the communication is made by fair links with the following properties: if the client and the recipient of a message are both correct then (1) if a message is sent infinitely often to a correct receiver then it is received infinitely often by that receiver; (2) there exists some delay T such that if a message is retransmitted infinitely often to a correct receiver according to some schedule from time t_0, then the receiver receives the message at least once before time $t_0 + T$; and (3) the message is not modified in the channel [6] [20]. This assumption appears reasonable in practice, since the DSR is running in a separated synchronous LAN and can only fail by crash, based on the VMM isolation.

3.1 Properties

The problem of atomic multicast, or total order reliable multicast, is the problem of delivering the same messages in the same order to all processes of a system. The definition in byzantine context can be seen as the following properties [7]:

Validity - If a correct process multicasts a message M, then some correct process eventually delivers M.
Agreement - If a correct process delivers a message M, then all correct processes eventually deliver M.
Integrity - For any message M, every correct process p delivers M at most once, and if sender(M) is correct then M was previously multicast by sender(M).
Total order - If two correct processes deliver two messages M1 and M2 then both processes deliver the two messages in the same order.

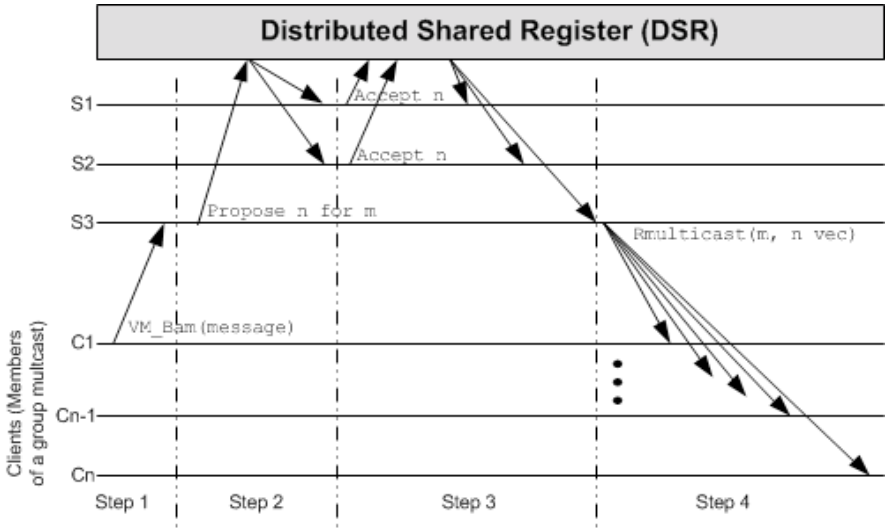


Fig. 2 Atomic multicast flow

4 Vbam algorithm

Discussion: The process needs only one server as sequencer. This server is responsible for proposing orders for client messages. Other servers are replicas of the service. Initially, the sequencer is the process which the id is zero. The sequencership change happens every time most servers ($f + 1$) agree to be necessary. As in others byzantine fault tolerant (BFT) system [5], [6], [15], to deal with the problem of a malicious server p_j, that would discard a message. Therefore the client p_i waits for receiving its own message with an order number for a T_{resend} time. After this T_{resend} the client send its message to all servers. A correct server when receives the client message, and it is not the sequencer, asks for a sequencer change. If $f + 1$ correct servers ask for a sequencer change, then it will be performed and the protocol makes progress. However, the payload system is assumed to be asynchronous, so there are no bounds on communication delays, and it is not

possible to define an ”ideal” value for T_{resend}. Correia [15], shows that the value of T_{resend} involves a tradeoff: if too high, the client can take long to have the message ordered; if too low, the client can resend the message without necessity. The value should be selected taking this tradeoff into account. If the command is resent without need, the duplicates are discarded by the system.

This section offers a deeper description of the algorithm. The sequence of operations of the algorithm is presented in the sequencer and destination nodes. It is first considered the normal case operation and, following, the execution where faults do exist. The flowchart of the normal case operation can be seen in the Figure 2. For clarity of presentation, we consider a single group multicast.

4.1 Normal operation

step 1) The process starts when some client c_i sends to the sequencer an ORDER message $\langle ORDER,m,t,v\rangle_{\sigma s_i}$ with the message m included. The field t is the timestamp of the message content to ensure one-time order semantics, in other words, servers will not order the message if t is not bigger than t - *1* for c_i. This politics prevents multiple ordering of a single message. The field v is vector that takes a MAC per server, each obtained with the key shared between the client and that server. Therefore, each server can test the integrity of the message by checking if its MAC is valid, and discard the message otherwise. In case the message has already been ordered, the server resends it to the client.

step 2) After verifying that MAC in v is correct and the timestamp is valid for the client's message, the sequencer generates a PROPOSE message as $\langle PROPOSE,n,o,mac\rangle_{\sigma p}$, where o is the original message, n the ordering number for it and *mac* is the message authentication code for m. The DSR automatically input in the proposal message the ID of the sequencer. The server will expect for the acceptance of the message, i.e., f processes agreeing with the proposal. Then the server, accept this order and saves it in the atomic buffer. All messages sent in the DSR will be delivered if the sender and the receiver are not crashed, as we have discussed in 4.

step 3) By receiving a proposal, the server s_k validates it, meaning that (i) s_k verifies, using the MAC in v, if the content of the message m is correct and (ii) verifies if there is no other proposal accepted before that with the same sequence number n. Then the proposal is accepted by s_k that writes an ACCEPT message on it's reserved space of the register. The message format is $\langle ACCEPT,n,h_m,mac\rangle_{\sigma p}$ and it contains the hash of the client's message h_m, the ordering number n to m and a message authentication code *mac* for m. After writing the accept message, the process waits for f - *1* acceptance messages to save it in the atomic buffer.

step 4) The sequencer reliably multicast the message with the order and the mac vector from at least f + *1* different servers that accepted it. After receiving

and validating the vector the clients finally accepts it and delivers in the proposed order.

4.2 Faulty operation

The faulty operation implies that a change will happen, therefore here is a brief explanation of how it develops.

Sequencer change: During the system configuration, all servers receive an identification number. This numbers are sequential and start at zero. All servers can recognize the identifier i of the sequencer and the total number n of servers in the system. When $f + 1$ correct servers suspect of the current sequencer they simply define $i = i + 1$ as the next sequencer if $i < n$, otherwise $i = 0$.

When validating a proposal the server s_k verifies if the message's content is correct using its MAC in the vector v and if the proposal is correct based on the earlier accepted proposals. If the message, for some of the reasons above, is not valid, then s_k will ask for a sequencer change.

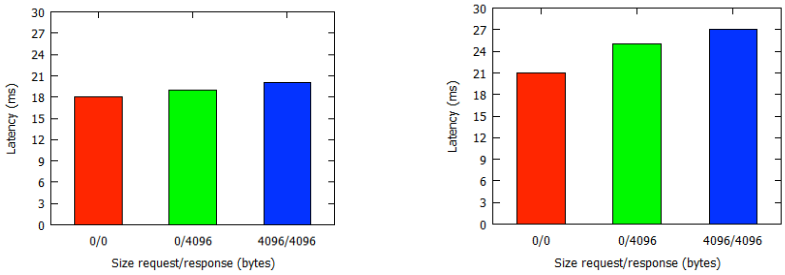
a) Correct servers can entry in the faulty operation by the two following ways:
   i) When the server s_k receives a change message e, but does not suspects the server s_s yet, the process just stores e in its local buffer.
   ii) When the process s_k suspects of the server s_s about a single message m, then s_k writes in the DSR and in it's own buffer a new message $\langle CHANGE, sid, h_m, S_s \rangle_{\sigma p}$ that contains sid as it's own identifier, the hash of the message h_m and s_s as the id of the server for which it suspects.
b) The server s_k starts a search in its buffer trying to find $f + 1$ change messages that relate to m and the server s_s. In case s_k finds $f + 1$ (including server s_k) different sid to the same propose message, then the server changes the sequencer to $s_s + 1$. If s_s is its own id then, based on the last accepted messages, it restart the ordering.

With the new sequencer the protocol make progress as in normal case operation.

5 Implementation and evaluation

The algorithms were implemented using the programming language Java, JDK 1.6.0, and executed using JVM Sun. The communication channels were implemented using TCP sockets from API NIO. The operational systems of the virtual machine hosts used were MacOSX Lion and Ubuntu 12.04. On virtual machines themselves, we used Ubuntu 12.04, Ubuntu 12.10 and Debian 6 Stable, being VirtualBox the VMM of choice. The adopted evaluation metric is latency, since this is one of the metrics most widely used to evaluate computational systems and that it represents the efficiency of the system in a very simple way [5], [6], [7], [8].

The values were obtained through micro-benchmarks in different loads. The latency was obtained measuring the round-trip time, that is, we measured the time between sending and receiving a group of messages. The reasoning behind the use of micro-benchmarks is to properly measure the algorithm without external influences. In order to gauge the protocol's capacity, we ran it with different message sizes.



(a) Latency - Normal Case (b) Latency with faults

Fig. 3 Performance Evaluation

To evaluate the algorithm's performance in the absence of faults, we ran it in normal conditions and sent 10.000 messages by a single client with three different loads: 0/0 kb, 0/4 kb and 4/4 kb. With those we have: an empty message and an empty ordered message, an empty message and a 4kb ordered message and a 4kb message with an 4kb ordered message. To evaluate the algorithm in a faulty environment, we conducted the experiment with $f = 1$ faulty server. In 3(a) and 3(b) we demonstrate the latency to each different load. The latency was obtained by the average from all total order multicasts. As we can observe, the latency has minimal variations between different loads.

We show the comparative data between our approach and the state of art for atomic multicast on the table 1. All the numbers consider non-faulty execution. Our approach benefits are visible as number of communication steps and resilience since we consider systems subject to byzantine faults.

6 Conclusion

By exploring the use of the Distributed Shared Register and virtualization techniques, we have managed to propose a simple inviolable LAN that supports our BFT atomic multicast. It was showed that it is possible to implement a reliable consensus service with only $2f + 1$ servers using common technologies, as is virtualization and data sharing abstractions. The virtualization technology is widely used and can provide a good isolation between the servers and the external world, and the use of the DSR makes it easy to maintain the protocol progress.

References

[1] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," ACM Computing Surveys (CSUR), vol. 36, no. 4, pp. 372–421, 2004.

[2] L. Rodrigues, P. Veríssimo, and A. Casimiro, "Using atomic broadcast to implement a posteriori agreement for clock synchronization," in Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on. IEEE, 1993, pp. 115–124.

[3] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," Knowledge and Data Engineering, IEEE Transactions on, vol. 15, no. 4, pp. 1018–1032, 2003.

[4] A. N. Bessani, J. da Silva Fraga, and L. C. Lung, "Bts: a byzantine fault-tolerant tuple space," in Proceedings of the 2006 ACM symposium on Applied computing, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 429–433.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," ACM Transactions on Computer Systems (TOCS), vol. 20, no. 4, pp. 398–461, 2002.

[6] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 253–267, 2003.

[7] M. Correia, N. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures," The Computer Journal, vol. 49, no. 1, pp. 82–96, 2006.

[8] F. Favarim, J. Fraga, L. C. Lung, M. Correia, and J. Santos, "Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids," in Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on, may 2007, pp. 403 –411.

[9] R. Ekwall, A. Schiper, and P. Urbán, "Token-based atomic broadcast using unreliable failure detectors," in Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on. IEEE, 2004, pp. 52–65.

[10] M. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in Proceedings of the 2nd ACM Conference on Computer and Communications Security. ACM, 1994, pp. 68–80.

[11] R. Guerraoui and A. Schiper, "The generic consensus service," Software Engineering, IEEE Transactions on, vol. 27, no. 1, pp. 29–41, 2001.

[12] G. Pieri, J. da Silva Fraga, and L. Lung, "Consensus service to solve agreement problems," in Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on. IEEE, 2010, pp. 267–274.

[13] M. Correia, P. Veríssimo, and N. Neves, "The design of a cots real-time distributed security kernel," Dependable Computing EDCC-4, pp. 634–638, 2002.

[14] P. Veríssimo, "Travelling through wormholes: a new look at distributed systems models," ACM SIGACT News, vol. 37, no. 1, pp. 66–81, 2006.

[15] M. Correia, N. Neves, and P. Veríssimo, "How to tolerate half less one byzantine nodes in practical distributed systems," in Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on. IEEE, 2004, pp. 174– 183.

[16] R. Guerraoui and L. Rodrigues, Introduction to reliable distributed programming. Springer-Verlag New York Inc, 2006.

[17] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no. 3, pp. 382–401, 1982.

[18] R. Rodrigues, M. Castro, and B. Liskov, "Base: Using abstraction to improve fault tolerance," ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 15–28, 2001.

[19] A. Menezes, P. Van Oorschot, and S. Vanstone, Handbook of applied cryptography. CRC, 1996.

[20] M. S. Wangham, L. C. Lung, C. M. Westphall, and J. Fraga, "Integrating SSL to the JACOWEB security framework: Project and Implementation," in IM'01, 2001, pp. 779–792.