BAMcast - Byzantine Fault-Tolerant Consensus Service for Atomic Multicast in Large-scale Networks

Marcelo Ribeiro Xavier Silva, Lau Cheuk Lung, Leandro Quibem Magnabosco, Luciana de Oliveira Rech
Computer Science - Federal University of Santa Catarina - Florianópolis, Brazil
marcelo.r.x.s@posgrad.ufsc.br lau.lung@inf.ufsc.br leandroqm@gmail.com luciana.rech@inf.ufsc.br

***Abstract*—This paper presents a BFT Atomic Multicast Protocol (BAMcast) for Wide Area Networks. Our algorithm manages to implement a reliable consensus service with only $2f + 1$ replicas using only common technologies, such as virtualization and data sharing abstractions. In order to achieve this goals, we chose to adopt a hybrid model, which means it has different assumptions between components regarding synchrony, and two different networks, a payload network and a separated network where message ordering happens.**

I. Introduction

Distributed systems often need to guarantee that messages are delivered to either all processes or to none at all. Some problems also need that the messages are delivered in the same order to all processes, it is know as the atomic (or total order) multicast problem. Atomic multicast has many important applications such as distributed shared memory, database replication, cooperative writing, clock synchronization [1], [2], [3] and are the basis of the state machine approach, the main component of many fault-tolerant systems [4], [5], [6].

The literature on total order multicast is vast with many approaches and algorithms have been proposed to solve this problem. In most cases, these algorithms consider system models subject only to crash faults [7], [8], [9]. Very few addresses also byzantine/arbitrary faults [10], [5]. 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]

Some works have shown that is possible to achieve atomic multicast in crash-only systems that assume the processes to be connected by an wide area network [13], [14], [15]. Scattering the processes geographically reduces the occurrence of failures caused by natural disasters, power outages and physical attacks [16].

We presente BAMcast, a byzantine fault tolerant consensus service for atomic multicast messages in a wide area network (WAN). We propose a consensus service with a hybrid system model that tolerates $f = \lfloor \frac{(n-1)}{2} \rfloor$ faulty servers. A hybrid system model implies that there exists variation, from component to component, on the assumptions of synchrony and presence/severity of failures [17], [18], [19]. In our architecture we use a payload network for the client-consensus service communication and a tamperproof network where the consensus service servers execute the ordination. In our proposal we contribute with an improvement to make the consensus service [11] byzantine fault tolerant with low resilience (only $2f + 1$ servers) and available for wide area networks, making use of some techniques as rotating the leader [16].

II. Atomic multicast

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 [5]:

AB1) Validity - If a correct process multicasts a message M, then some correct process eventually delivers M.
AB2) Agreement - If a correct process delivers a message M, then all correct processes eventually deliver M.
AB3) 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 $sender(M)$.
AB4) Total order - If two correct processes deliver two messages $M1$ and $M2$ then both processes deliver the two messages in the same order

Following these properties we show an atomic multicast protocol based on virtualization technology and emulated shared memory [20].

III. Related work

The atomic multicast problem has been widely addressed in the past decades [21], [1], [10], [22], [13], [8], [2], [9], [5]. In the majority, the approaches consider that the processes can only crash, i.e., not acting in arbitrary/byzantine manner.

In 1993 Berman and Bharali presented the Quick-a [21] as a solution for the atomic multicast problem in byzantine systems. In their algorithm the processes maintain a round number that serves as a timestamp for the messages broadcasted. The processes execute a sequence of a randomized binary consensus to decide on the round in which messages will be delivered.

Rampart, for reliable and atomic multicast messages in systems with byzantine faults, is presented in [10]. 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, that 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.

In 1995 Cristian et al [22] presented HAS, that is a collection of total order broadcast algorithms. It assumes a synchronous model with ϵ-synchronized clocks. Each message broadcasted has a timestamp T of its emission that is used for the processes delivery in conjunction with the Δ value (deliveries are made at time $T + \Delta$ based in the processes local clocks). The Δ value considers the topology of the network, the number of failures tolerated, and the maximum clock drift ϵ.

Rodrigues et al proposed in [8] an atomic multicast for crash-only environments divided into groups where the processes to which a message m is multicast, exchange the timestamp they assigned to m, and, once they receive this timestamp from a majority of processes of each group, they propose the maximum value received to consensus. Because consensus is run among the addressees of a message and can thus span multiple groups, this algorithm is not well-suited for wide area networks, even though it is scalable.

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 separate the consensus from the agreement problem to be solved. The system requires a perfect failure detector (that basis the consensus server) and the resilience varies according to the need (and it implies in a trade-off with performance).

In 2006 Correia and Veríssimo showed a transformation from consensus to atomic broadcast [5]. The system model presented assume a byzantine environment in which up to $f = \lfloor \frac{(n-1)}{3} \rfloor$ 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 m_i to the clients set. Upon receiving the message m_i, each client sends a proposal to the generic consensus service 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.

IV. System Model and Architecture

The system model is hybrid [18], which is where assumptions of synchrony and presence/severity of failures vary from component to component [17], [18]. 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, $C = \{c_1, c_2, c_3, ...\}$ is a set that contains a finite number of client processes and $S = \{s_1, s_2, s_3, ..., s_n\}$ representing a set of servers with n elements that compound the consensus service. We assume that both, clients and servers, are connected by a Wide Area Network (WAN), which means they can be geographically scattered.

In the Figure 1, each consensus service replica 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 $f \leq [\frac{n-1}{2}]$ servers incurring failure based on its specifications, presenting byzantine faults [23]: 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) [24].

The system has two distinct networks, the payload and the controlled networks (see Figure 1). The former is asynchronous and is used for application data transfers. There are no assumptions based on time on the payload network and it is used for client-server communication. The later, used for server-server communication, is a controlled network composed by physical machines, where is implemented a Distributed Shared Register (DSR) [20] (see subsection IV-A). The consensus service uses the DSR to execute the crucial parts of the consensus protocol. The DSR has the following properties:

- it has a finite and known number of members;

Table I: Comparison on evaluated atomic multicast protocols properties

	Rampart [10]	Rodrigues et al [8]	Guerraoui and Schiper [11]	Correia and Veríssimo [5]	Pieri et al [12]	BAMcast
Resilience for atomic multicast	$3f+1$	$2f+n_g$	-	$3f+1$	$3f_c+1+2f_s+1$	$2f+1$
Communication Steps	6	6	5	-	5	4
Messages exchanged	6n - 6	$3(d-1)^2+4(d-1)+(d-1)^2$	$3n_c+2n_s-3$	$18n^2+13n+1+16n^2f+10nf$	$2(Ns^2+3Nc-Ns-1)$	${N_s}^2-N_s+N_c+1$
Large scale	no	yes	no	no	no	yes
Tolerated faults	byzantine	crash	crash	byzantine	byzantine	byzantine

- it is assumed to be secure, i.e., resistant to any possible attacks; it can only fail by crashing;
- it is capable of executing certain operations with a bounded delay;
- it 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 BLACKLIST messages. All the servers can read all the register space, no matter who holds 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 [19]. 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 [25], [4]. The processes have to share symmetric keys in order to use MACs. In this paper 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 [25]. This issue is out of the scope of this paper.

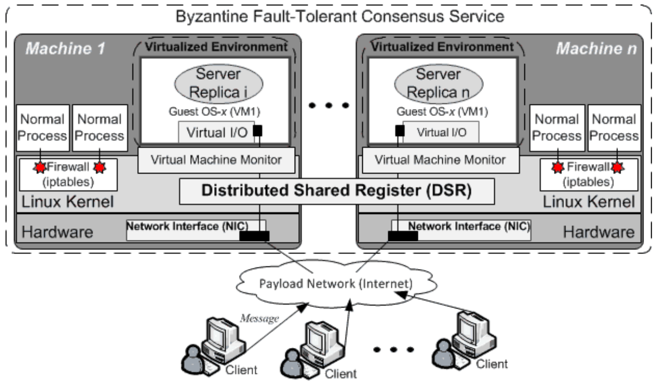

Figure 1: BAMcast Architecture Overview

We assume that only the physical machine can, in fact, connect to the controlled network used for the register. That is, the DSR is accessible only by the physical machines that host virtual machines 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 network (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 can not 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.

A. Emulated Shared Memory and the Distributed Shared Register

Emulated shared memory is an abstraction of registers from a set of processes that communicate through message passing [20]. This definition is really attractive, since it allows the shared memory to be built using any technology for sharing memory. A shared memory, emulated or not, can be seen as an array of shared registers. We consider here the definition under the programmer point of view. The type of the shared register specifies what operations can be performed and the values returned by the operation [20]. The most common types are read/write register. The operations of a register are invoked by the system processes to exchange information. We created the Distributed Shared Register (DSR), an emulated shared memory based on message passing over a controlled network and making use of local files. We assume the controlled network 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 Distributed Shared Register 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 return the last value written. To ensure this properties, in each replica 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 [20]. The first file is the replica space, and no other replica can write on it. The second one is the other replicas 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) BLACKLIST. 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. This assumption appears reasonable in practice, since the DSR is running in a separated synchronous network and can only fail by crash, based on the VMM isolation.

V. BAMCAST ALGORITHM

Protocol overview: We borrowed the asynchronous views and the rotating primary from [16]. This means that the clients send a message to the nearest server in the consensus service group, reducing the end-to-end latency. All messages are ordered in parallel, but the reliable multicast only occur in the view where the server is the leader (or primary). The first leader is the lowest id (zero) and the next leader (and view) is the last one plus one. If the new leader already has accepted messages they are delivered. When a server has no messages to reliably multicast, it just sends a skip message as in [16]. As in other byzantine fault tolerant (BFT) systems [4], [19], to deal with the problem of a malicious server p_j, that would discard a message, the client p_i waits for receiving its own message in ordered for a T_{resend} time. After this T_{resend} the client picks another server and resends the message. 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 [19], 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.

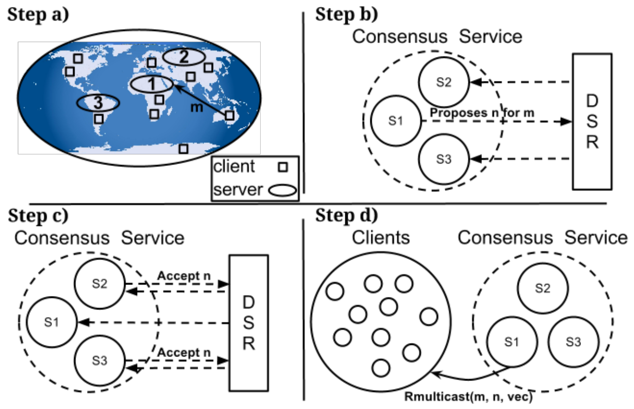

Figure 2: Atomic multicast flow

A. Normal operation

a) The process starts when some client c_i sends to some server an ORDER message $\langle ORDER, m, t, v\rangle_{\sigma ci}$ 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.

b) After verifying that MAC in v is correct and the timestamp is valid for the client's message, the server generates a PROPOSE message as $\langle PROPOSE, n, o, mac\rangle_{\sigma si}$, where o is the original message, n the ordering number for it and mac is the message authentication code for m. The DSR automatically added to the proposal message the ID of the server. The server will expect for the acceptance of the message, i.e., f processes agreeing with the proposal. Then the server accepts this order and saves it in the atomic buffer. This behavior can be observed on the algorithm 1. All messages sent in the DSR will be delivered if the sender and the receiver are not crashed, as we have discussed in IV.

c) 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 its reserved space of the register. The message format is $\langle ACCEPT, n, h_m, mac\rangle_{\sigma si}$ and it contains the hash

```
Algorithm 1: Proposal algorithm
Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided
Variables:
accepted : int // counter of acceptance for some ordering
1  upon receive $\langle ORDER, m, t_j, v\rangle_{\sigma ci}$ from client
2  if alreadyProposed( m ) then
3      rmulticast( getReply( m ) from buffer );
4      return;
5  end
6  if $t_j \leq t_{j-1}$ for $c_i$ OR isWrong( v ) then
7      return;
8  end
9  write( $\langle PROPOSE, n, \langle ORDER, m, t_j, v\rangle_{\sigma ci}, mac\rangle_{\sigma si}$ ) into DSR;
10 accepted = waitForAcceptance( T );
11 if accepted $\geq f$ then
12     bufferize( $\langle REPLY, m, n, svec\rangle_{\sigma si}$ );
13 end
14 return;
```

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. This behavior can be observed on the algorithm 2.

d) The messages are kept in the buffer until the view for where they were accepted became the actual one. When it happens, the primary server reliably multicast for the clients the message $\langle REPLY, m, n, svec\rangle_{\sigma si}$ with the original message m, its order number n and the mac vector $svec$ 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.

```
Algorithm 2: Acceptance algorithm
Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided.
Variables:
accepted : int // counter of acceptance for some ordering
1  upon read $\langle PROPOSE, n, \langle ORDER, m, t, v\rangle_{\sigma ci}, mac\rangle_{\sigma ss}$ from
   DSR
2  if isValid( m, v ) and isValid( n ) then
3      write( $\langle ACCEPT, n, h_m, mac\rangle_{\sigma si}$ ) into DSR;
4      accepted = waitForAcceptance( T );
5      if accepted $\geq f-1$ then
6          bufferize( $\langle REPLY, m, n, svec\rangle_{\sigma si}$ );
7      end
8  else
9      write( $\langle BLACKLIST, h_m, s_s\rangle_{\sigma si}$ ) into DSR and bufferize;
10 end
11 return;
```

1) Faulty operation: The faulty operation implies that a server will be blacklisted, therefore here is a brief explanation of how it develops.

Blacklisting: During the system configuration, all servers receive an identification number. This numbers are sequential and start at zero. All servers know the total number of servers in the consensus service . When $f+1$ correct servers suspect that some server is malicious they simply blacklist the server. The blacklisted server can not be leader, but it still participates in the algorithm. If the server is the leader of the actual round the others servers change the view defining $i = i+1$ as the next leader if $i < n$, otherwise $i = 0$. The blacklist has size f, since it is the maximum tolerated faults in the system.

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 blacklisting the proponent server.

a) A correct server will start operating in faulty mode by one of the two following ways:
   1) When the server s_k receives a blacklist message e, but does not suspects the server s_s yet, the process just stores e in its local buffer, in order to use this message if needed in the near future.
   2) 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 its own buffer a new message $\langle BLACKLIST, h_m, s_s\rangle_{\sigma s}$ that contains sid as its own identifier automatically filled by the DSR, 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 $f+1$ blacklist 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 puts s_s in the blacklist. If s_s is the leader of the actual view, then the view is changed and the servers buffer only the messages that were accepted by majority.
c) With the new view the protocol make progress as in normal case operation.

B. *Correctness proof*

The atomic multicast protocol is correct if it satisfies the properties AB1-AB4 discussed in II.

Validity and agreement proof. A correct client unicasts a message to some server. If the server is honest, then it will write down the message in the DSR and, as we discussed in IV-A, it will reliably multicasts the message. This server behavior can be seen in the line 9 in algorithm 1. After receiving a proposal, each server deliberate if it is valid and when the majority agrees on it, then every correct server will store in the buffer to reliably multicast it if needed, as can be seen in lines 10-13 in algorithm 1 and 2-8 in algorithm 2. In case the server that proposed is corrupt, the client will resend the message to the next server in its list after T_{resend}, and if, the new server is honest, it will propose an order for the client message. It will occur until an honest server receive the client message. The correct clients will deliver the message in the order proposed by the consensus service.

Integrity proof. The line 2 of the algorithm 2 requires the servers to verify, using the MAC vector, if the message was sent by $client(m)$ and as discussed in V-A the timestamp (or message ID) t of m must respect the rule $t > t-1$ or has been already proposed (lines 2-8 of algorithm 1), otherwise the message m is automatically discarded. The correct clients will deliver the message just once.

Total order proof. Any correct server reliably multicasts messages only after an execution of a consensus (lines 9-14 and 2-11, respectively from algorithms 1 and 2). The messages multicasted are those that were accepted by at least $f+1$ servers and the delivery order is deterministic (9 of algorithm 1). Therefore, the servers reliably multicast the messages with the vector proving that at least $f+1$ servers accepted the order and the correct clients will deliver messages it in the same order.

VI. Conclusion

By exploring the use of the Distributed Shared Register and virtualization techniques, we have managed to propose a simple inviolable network that supports our BFT atomic multicast. It was showed that it is possible to implement a reliable consensus service with only $2f+1$ replicas using common technologies, as is virtualization and data sharing abstractions. We proved that is possible to make atomic multicast in wide area networks with clients and servers geographically scattered. The virtualization technology is widely used and can provide a good isolation between the replicas and the external world, and the use of the Distributed Shared Register makes it easy to maintain the protocol progress. Furthermore, the use of asynchronous views and rotating the primary made possible to use a wide area network to connect clients and servers.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable atomic multicast," in *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*. IEEE, 1998, pp. 840–847.

[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] L. Rodrigues, H. Fonseca, and P. Verissimo, "Totally ordered multicast in large-scale systems," in *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*. IEEE, 1996, pp. 503–510.

[14] A. Sousa, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. IEEE, 2002, pp. 190–199.

[15] P. Vicente and L. Rodrigues, "An indulgent uniform total order algorithm with optimistic delivery," in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. IEEE, 2002, pp. 92–101.

[16] G. Veronese, M. Correia, A. Bessani, and L. Lung, "Ebawa: Efficient byzantine agreement for wide-area networks," in *Proceedings of the 12th Symposium on High-Assurance Systems Engineering*. Citeseer, 2010, pp. 10–19.

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

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

[19] M. Correia, N. Neves, and P. Verissimo, "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.

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

[21] P. Berman and A. Bharali, "Quick atomic broadcast," *Distributed Algorithms*, pp. 189–203, 1993.

[22] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," *Information and Computation*, vol. 118, no. 1, p. 158, 1995.

[23] 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.

[24] 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.

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