Consensus Service to Solve Agreement Problems

Giovani Pieri, Joni da Silva Fraga
Departamento de Automação e Sistemas
Federal University of Santa Catarina
Florianópolis, Brazil
Email: {pieri, fraga}@das.ufsc.br

Lau Cheuk Lung
Departamento de Informática e Estatística
Federal University of Santa Catarina
Florianópolis, Brazil
Email: lau.lung@inf.ufsc.br

***Abstract*—This paper describes an extension of the Consensus Service proposed by Guerraoui and Schiper. The objective is to provide a standard way to implement agreement protocols resilient to Byzantine faults using an intrusion tolerant service built upon virtual machines technology. This is achieved through the implementation of a Generic Consensus Service (GCS). GCS separates specificities of different agreement problems from consensus in a clear way, using client-server interaction, allowing total independence between consensus protocols used and problem specific specializations. Besides that, the framework provides a set of properties and guarantees. It will be shown how the GCS works, its general properties and how it may be used to solve some agreement problems, for instance, reliable and atomic broadcast.**

***Index Terms*—Dependable Systems, Consensus, Intrusion tolerance, Distributed algorithm, Fault tolerance**

I. Introduction

Agreement problems, such as atomic commitment, group membership, and total order broadcast are at the heart of many distributed systems. These problems share one common characteristic, they require that each process after the completion of the protocol agrees on the outcome of the distributed computation that has taken place. Usually each one of these problems is tackled separately and solved by specialized algorithms. However, in [19] it was proposed that one of the simplest agreement problems, namely consensus, should be used as a paradigm to build distributed protocols. Besides, it has been shown that efficient protocols to solve agreement problems may be developed using consensus as a building block in Byzantine settings [9], [22].

In this paper we propose an extension of the Consensus Service [13] proposed by Guerraoui and Schiper to a Byzantine environment, and two consensus algorithms based on different trusted components. The objective of consensus service is to provide a intrusion-tolerant common infrastructure to agreement problems. The consensus service proposed in [13] is able to tolerate crash faults, we take it one step further and create an intrusion tolerant service. Note that, in our proposition, the clients of this service may suffer Byzantine failures.

Inside the consensus service each process has a role: initiator, client and server. In order to solve a problem, the initiator process decides that a consensus must be started and reliably multicast a message to the clients. The clients receive this message and compute a proposal accordingly. This proposal is sent to the servers responsible for executing the consensus. After the servers reach a consensus they reply the decision value to the clients.

Despite similarities in agreement problems, each one has its own specificities. To deal with them, Guerraoui and Schiper use what is called a consensus filter. The consensus filter is responsible for calculating a proposal to be used in the consensus protocol, given the data received from clients, so that the protocol outcome satisfies the given agreement problem.

Recently there has been a trend trying to achieve greater resilience [5], [6], [8], [17], [18] despite the existence of Byzantine faults by trusting in part of the system. With adequate reliable components [6], [15], [16] it is possible to tolerate up to f out of $n = 2f+1$ faulty processes and decrease the number of steps required by an agreement protocol. One of the problems with this approach is that it requires a trusted component, therefore some kind of measure has to be taken to guarantee its inviolability. The generic consensus service allows us to take advantage of this new trend while not having to make assumptions on every node of the system. Due to the complete isolation between the execution of the consensus protocol from the agreement problem, only those who execute the consensus are augmented with trusted components.

The rest of the paper is structured as follows. Section III defines the system model, our assumptions, processes behaviors and the architecture. Section IV presents the generic consensus service, the general algorithms and the internal working of the framework. Section VI illustrates the use of the generic consensus service on two agreement problems: reliable broadcast and atomic broadcast. Section VII presents algorithms to solve consensus through specially designed reliable component. Section VIII we show some results and comparisons. Section IX concludes the paper.

II. Related Works

This work is based on consensus service introduced by Guerraoui and Schiper [13] in a crash-only environment. This consensus service service aims to separate the resolution of consensus from the specificities of each agreement problem. It provides a framework upon which crash-tolerant agreement protocols may be built by defining a consensus filter, responsible for the reduction of an agreement problem to consensus.

Several works proposed the use of consensus protocols as basis upon which agreement problems could be built [19].

In a Byzantine environment, it has been shown that efficient consensus based algorithms to solve problems such as vector consensus and atomic broadcast may be built on top of consensus algorithms [9].

The use of virtual machines and virtualization technology to deliver intrusion tolerant services have been researched recently. The VM-FIT architecture [17] is one of the first known proposal of the use of virtualization to achieve fault-tolerance. In this work, several virtual machines run on top of an only physical system, reducing the implementation cost of the system. Up until now hardware redundancy was obligatory to leverage fault tolerance support.

In LBFT1 and LBFT2 [5] it is proposed an approach in which a virtual system assumes the role of a trusted component. This trusted component orders the requests made by clients and delivers them to the service VMs in order to be executed.

Wormholes abstractions were introduced in [7]. It proposes the reduction of implementation costs of a distributed fault tolerant algorithm through the use of a trusted component called TTCB which gives access to a private system which has different synchrony and fault models. Using this system it is possible to implement reliable broadcast among $n \geq f+1$ participants being up to f Byzantine.

Several other proposals for trusted components have been made in recent years. In the A2M [6], it is proposed a trusted log service. In the TrInc [16], a minimal trusted component is built on top of a Trusted Platform Module [15], [21], easing the system deployment.

III. System Model

We consider an asynchronous distributed system composed by a finite set of processes Π ($\Pi = \{p_1, p_2, \ldots, p_n\}$). These processes may be classified into correct or faulty during the execution of the algorithm. A correct process will always follow its algorithmic specification. Faulty process may suffer Byzantine failures, i.e., they behave arbitrarily. A faulty process may omit or send invalid messages and are not required to follow the algorithm. Besides that, faulty processes may coordinate in order to stall the progress of the distributed system or lead it to an invalid state.

Any two processes are connected through a reliable authenticated asynchronous channel. These properties guarantees that every sent message is eventually delivered, the sender of a message is known and the message integrity is guaranteed. Although we do not make any synchrony assumption in order to guarantee safety properties of the algorithm, we must assume eventual synchrony to ensure liveness properties. If the system was completely asynchronous it would be impossible to solve consensus, according to the FLP condition [12].

A. Process Roles

As consensus service in [13], the generic consensus service (GCS) proposed here is composed by n_s server, n_c client and n_i initiator processes. The servers are running in separated nodes than clients and initiators. The sets of clients and initiators may intersect. Client processes are the ones that have an agreement problem that needs to be solved in the context of an distributed application algorithm AP. By solving a consensus problem, server processes are responsible for collecting proposals from clients, creating and deciding a valid vector for replying it to the clients. The initiator processes start a consensus instance where clients will query servers. As the name implies, the GCS is general and server processes are agnostic to the proposals' and results' meanings. It is responsibility of each client to convert the result received from the servers into a useful value to solve the particular agreement problem in which the clients are interested in. This conversion is done through the concept of a consensus filter (CF). This CF is composed by a deterministic function, called `Result`, that takes the decision calculated by the servers and produces the value that will be consumed by the client.

During the execution of a consensus instance, there are at most f_s faulty server processes, f_c faulty client processes. In order to support the presence of f_c faulty clients, our algorithms require that $n_c = 3f_c + 1$. The resilience of the set of server processes is dictated by the consensus protocol used. Through the adoption of a reliable component, it will be required only $n_s = 2f_s + 1$ processes to support f_s faulty servers.

The set of initiators may be composed by any number of processes. These processes must ensure that all correct clients eventually initiate consensus instances.

B. Virtual Machines

We will employ virtualization technology in part of the system. The virtual machines run on top of a Virtual Machine Monitor (VMM). Our objective in using VMs is to isolate the malicious aspects from the crash fault tolerance. The faults that may be experienced by a VM are unrestricted, while the VMM only suffers crash faults.

The VMM provides to all VMs access to trusted components, presented in section VII. These components make some services available to the VMs, which prevents equivocation [6] of faulty processes. This leads to more resilient algorithms able to tolerate f out of $n = 2f + 1$ processes instead of $3f + 1$ required in a traditional Byzantine system model.

The host operating system may have security flaws (vulnerabilities), however, they can not be exploited through the network. This claim is feasible because we assume that the host operating system is inaccessible. This may be easily implemented in a real system using a firewall system or disabling/removing the network drivers of the host operating system. In our model, only the VMs have direct access to the network interfaces.

We assume that VMM provides isolation between VMs, i.e., faults in one VM can not influence another one. This is assumption is guaranteed by virtualization technology (e.g. VirtualBox, XEN, VMware, VirtualPC, etc).

C. Communication Primitives

A process may use 2 different communication protocols for broadcasting a message to a given set of processes:

`Rmulticast` and `Multisend`. `Multisend` is composed by two operations `Multisend()` and `Receive()`. By using Multisend, if the sender is faulty, then no assumption can be made about the reception in the destination processes. However, if the sender is correct then every correct process receives the same message and delivers it through `Receive()`. `Multisend` is built using reliable point-to-point communication channels.

The Rmulticast protocol ensures a consistent behavior even when the message sender is faulty. A Rmulticast is composed by two operations `Rmulticast()` and `Rdeliver()`. A message sent by a correct process through `Rmulticast()` will eventually be `Rdelivered` by all correct processes. If the sender is faulty then correct processes will deliver the same message or do not deliver it at all. Reliable multicast and its primitives are thoroughly studied in the literature in different model systems [1], [4], [14].

IV. The Generic Consensus Service Problem

GCS modularizes the implementation of agreement problems in Byzantine settings. This is done by separating the consensus protocol execution from the agreement problem that must be solved by the clients. The GCS is presented as a framework to clients and initiators. These processes implement some specific functions and procedures that are called by the application for solving an agreement problem using GCS. For instance, the computation of the consensus instance identification, proposal and processing of the results are all problem-specific and left open to specializations.

In the GCS model, the roles of initiators and clients may overlap. The initiator starts a consensus instance by Rmulticast a SYNC message to the clients. Then, as consequence, the clients will calculate and send messages with their proposals to the servers, using Rmultisend. These proposals are specific to the problem that the clients must solve. The servers collect the proposals, run a consensus protocol to reach agreement and reply to the clients an agreed value.

In our approach, two modes of relationships among initiators and clients are distinguished. In the first mode, initiators and clients are separated and a consensus instance is started by a Rmulticast of SYNC message. In the second one, the set of initiators and clients are exactly the same. In this last case, it is possible to initiate an agreement without using reliable broadcast. However, it must ensure that all correct processes send a proposal regarding a specific consensus instance to server processes. Otherwise, the servers will not be able to create a valid response to the clients.

Is is important to highlight that a malicious initiator may perform attacks. However, these attacks are limited by the use of a reliable multicast to send the synchronization messages. A malicious initiator may only attempt to start an invalid consensus instance or exhaust identification values space. The first attack is application dependent and should be dealt with according to the specific agreement protocol. The former may be solved through several techniques such as low and high watermarks [3].

1: **procedure** RequireStart
2: $idC \leftarrow$ CalculateId
3: $Rmulticast\langle \text{SYNC}; idC\rangle$ to *clients*
4: **end procedure**
5: **procedure** RequireLocalStart(idC)
6: **send** $\langle \text{SYNC}; idC\rangle$ **to** p_i
7: **end procedure**

Fig. 1. Algorithm of initiator p_i

Malicious clients may launch a denial of service attacks, sending messages with invalid proposals to servers. However, servers should be able to resume its services after the attack is finished.

As shown in [10], [11] in order to reduce another agreement problems, as atomic broadcast to consensus for instance, it is necessary to define a stronger validity property to the classic agreement definition for crash failures. In this paper, we are founded on the definition of vector validity.The servers collect proposals and agree a vector that satisfies the vector validity property.

V. Algorithmic Basis of GCS

As stated above, the GCS acts as a framework. In this section we will present the algorithms that form the core of this framework.

Correctness proofs were developed to the algorithms presented, unfortunately due to space restrictions we will omit them. The proofs are available online at: http://www.inf.ufsc.br/~pieri/SGCProofs.pdf

Each one of the processes in the agreement problem to be solved obey an algorithm depending on the role they play in the GCS. The initiator processes follow the algorithm 1. This algorithm provides a procedure called `RequireAgreement` to initiate a consensus instance, whose identifier is calculated through a call to a procedure `CalculateId` that must be implemented according to the specific problem to be solved by GCS. If the client and initiator sets are equal, then the procedure `RequireLocalStart` may be used to start an agreement. This procedure forces the client who has executed it to send its proposal to the servers. Therefore, the GCS's specialization must ensure that all correct clients send their proposals to the servers. If this constraint is not met, the servers will not decide and will not reply the clients' request. The `RequireStart` satisfies this requirement because if one correct process receives a SYNC message, every other correct process will receive the same SYNC message and start the agreement.

GCS clients follow the algorithm 2. On receive a message SYNC from an initiator process, the client calculates its proposal and assign it to the variable $proposal$ (line 5). Then, this variable is sent to the GCS's server processes through a `Multisend()`, concatenated with the consensus instance identifier `idC` and a signature (line 8). The client waits the arrival of the replies from the servers. When the client receives $f_s + 1$ responses with the same decision vector from distinct servers (line 13), it applies the Consensus Filter (CF)

function `Result` to the vector (lines 14 and 15) and handles the control to the application through a call to the method `AgreementFinished`.

Init:
1: $proposal_i \leftarrow 0$; $sign_i \leftarrow 0$
2: $resp_i \leftarrow \perp$; $v \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
3: $received\_msgs_i^{idC} \leftarrow \emptyset$; $validMessage \leftarrow false$
4: **on *Rdelivery*** $\langle$SYNC; $idC\rangle$ **from** $Initiator$
5: $proposal_i \leftarrow$ CALCULATEPROPOSAL(idC)
6: $sign_i \leftarrow$ SIGN($\langle idC; proposal\rangle$)
7: $req_i \leftarrow \langle$PROPOSE; idC; $proposal$; $sign\rangle$
8: $MultiSend\ req_i\ to\ GCS$
9: **on receive** $\langle$DECIDE; idC; v; $sign_j\rangle$ **from** s_j
10: $validMessage \leftarrow$ VERIFIEDSIGNATURE($sign_j, s_j$)
11: **if** $validMessage$ **then**
12: $received\_msgs_i^{idC} \leftarrow received\_msgs_i^{idC} \cup \{v\}$
13: **if** $(\#_v received\_msgs_i^{idC} = f_s + 1)$ **then**
14: $result_i \leftarrow$ RESULT(v)
15: $AgreementFinished(idC; result_i)$
16: **end if**
17: **end if**

Fig. 2. Client process c_i from the GCS

The server processes follow the algorithm 3. Upon receipt of $n_c - f_c$ PROPOSE messages from clients proposing a value to the same consensus instance (line 10), the server start a consensus protocol (line 11). Then, the result of this protocol is relayed to the clients through the `Multisend` of a message $\langle$DECIDE; idC; $agreedVector\rangle$ (line 17).

The servers build a vector based on the received proposals. This vector is certified through a set of messages digitally signed by the clients. The consensus protocol used, actually, is a variation of the classical consensus problem named Certified Initial Value Consensus. This variation of consensus and an algorithm solving it in presence of Byzantine faults are introduced in [11]. This problem guarantees that the agreed value is a certified initial value.

In order to satisfy the needs of server processes and to guarantee the vector validity properties required, the following definition of certified vector is made:

Definition 5.1: A server process s_i considers a vector $vector_i^{idC}$ certified in respect of a consensus instance idC, if and only if:

1) $\#_\perp vector_i^{idC} \leq f_c$
2) $\forall j \in [1, n_c], \exists msg_j \in cert_i^{idC} : vector_i^{idC}[j] \neq \perp$
$\Rightarrow msg_j.proposal = vector_i^{idC}[j]$
$\wedge$VERIFIEDSIGNATURE($msg_i.sign, c_j$)
$\wedge msg_j.idC = idC$

The definition 5.1 states that certified vector contains at least $n_c - f_c$ clients proposals and is associated with a certificate composed by a set of digitally signed messages. Every proposal in the vector must have the respective sender's PROPOSAL message in the certificate.

VI. APPLICATIONS

In this section, two examples are presented of using GCS to solve agreement problems: Reliable Broadcast and Atomic Broadcasts. Despite of the presentation of just two agreement problems in this paper, it is by no means the only problems that may be tackled by the GCS. We have also defined algorithms to solve Strong Consensus, Vector Consensus, Group Membership, and a hybrid approach to Atomic Non Blocking Commit.

Init:
1: $cert_i^{idC} \leftarrow \emptyset$; $vector_i^{idC} \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
2: $agreedVector_i^{idC} \leftarrow \langle \perp; \perp; \ldots; \perp \rangle$
3: $resp\_sign_i \leftarrow 0$; $validMessage \leftarrow false$
4: **on receive** $\langle$PROPOSE; idC; $proposal_j$; $sign_j\rangle$ **from** c_j
5: $validMessage \leftarrow$ VERIFIEDSIGNATURE($sign_j, c_j$)
6: **if** $validMessage \wedge vector_i^{idC}[j] = \perp$ **then**
7: $vector_i^{idC}[j] \leftarrow proposal_j$
8: $prop_i^{idC} \leftarrow \{\langle$PROPOSE; idC; $proposal_j$; $sign_j\rangle\}$
9: $cert_i^{idC} \leftarrow cert_i^{idC} \cup prop_i^{idC}$
10: **if** $(\#_\perp vector_i^{idC} = f_c)$ **then**
11: PROPOSE($idC; vector_i^{idC}; cert_i^{idC}$)
12: **end if**
13: **end if**
14: **on decide** $\langle idC; agreedVector_i\rangle$
15: $resp\_sign_i \leftarrow$ SIGN($\langle idC; agreedVector_i\rangle$)
16: $msg_i \leftarrow \langle$DECIDE; idC; $agreedVector_i$; $resp\_sign_i\rangle$
17: **MultiSend** msg_i **to** *clients*

Fig. 3. Server process s_i from the GCS

Init:
1: $proposal_i^{idC} \leftarrow \perp$
2: **procedure** $R\_Broadcast(msg_i)$
3: $MultiSend\ \langle INITIAL, msg_i\rangle\ to\ clients$
4: **end procedure**
5: **on receive** $\langle INITIAL, msg_j\rangle$ **from** c_j
6: $idC \leftarrow msg_j.id$
7: $proposal_i^{idC} \leftarrow msg_j$
8: REQUIRELOCALSTART(idC)
9: **procedure** CALCULATEPROPOSAL(idC)
10: **return** $proposal_i^{idC}$
11: **end procedure**
12: **procedure** AGREEMENTFINISHED($idC, result$)
13: $R\_deliver(\langle idC, result_i\rangle)$
14: **end procedure**

Fig. 4. Client/initiator c_i of Reliable Broadcast

A. Reliable Broadcast

Reliable Broadcast (RB) is a group communication problem whose objective is to send a message to all processes belonging to a group. Two primitives define this problem: `RMulticast()` and `RDeliver()`. The reliable broadcast ensures that if a correct process `RMulticast()` a message then every correct process will deliver this same massage. A precise definition of validity, agreement and integrity properties for RB in Byzantine environments may be found in [2].

In the following we assume that, as in [2], message identifiers are formed by the concatenation of two components $msg_j.id = j|sn$, where j is the sender's identifier and sn is the messages serial number according to the sender. Algorithms 4 and 5 show the needed specializations for implementing a RB protocol using GCS."

For building a reliable broadcast protocol, the initiator and client sets are equal. The algorithm 4 describes the behavior of RB participant. It exposes a procedure `RBroadcast()` to the application that will be used to broadcast a message to the

```
1: function RESULT(vector)
2:     return MAJORITY(vector)
3: end function
```

Fig. 5. Result Function to Solve Reliable Broadcast

group. A broadcasted message is received by a process when procedure `RDeliver()` is invoked.

The algorithm explores the multisend's property that states that if the sender of a message is correct, then every correct process will eventually deliver it. When a correct process wants to broadcast a message it will `Multisend()` it. This message is received by every correct process that will propose it to the GCS servers. When the servers reach a consensus over the message identified by id, they will reply the clients with a vector. This vector goes through the Consensus Filter to the RB protocol (algorithm 5) that, in this case, will choose the most frequent message to be delivered. Afterwards, the agreed messages are `RDelivered()` to the application.

This algorithm respects all the properties required by the RB problem. If the sender is correct, then every correct client will receive the same message through the `MultiSend()`, then all correct processes will propose the same message to the GCS. As the GCS guarantees that there are at least $f_c{+}1 > f_c$ correct values in the agreed vector (vector validity), the majority function used in the consensus filter will return the message broadcasted by the client. If a message is sent by a faulty sender, if a correct process delivers it the GCS guarantees that all correct processes will deliver the same message.

B. Atomic Multicast

Atomic Multicast (AM) is a group communication protocol defined by two primitives: `A-Multicast()` and `A-Deliver()`. AM enforces the same properties as RB, but additionally requires ordering. Every correct process must deliver messages in the same order. As in the RB protocol, we assume that message identifiers are composed by two parts $msg_j.id = j|sn$ where j is the sender's identifier and sn a serial number.

The algorithm shown in figure 6 explicit the following behavior for an Atomic Multicast: when a given process wants to broadcast a message msg it will invoke the procedure `A-Multicast()`. This procedure will use a reliable broadcast to send these messages to all processes. When a message is RDelivered by the reliable multicast protocol, the atomic broadcast protocol will include this message in the set $messagesToDeliver$. At this point, the order of the messages must be specified. In order to do this, the client will query the GCS.

The AM algorithm advances in asynchronous rounds, in each one it will calculate a set of stable messages to be delivered. A message msg is considered stable if there are $f_c + 1$ clients reporting that msg must delivered. This guarantees that at least one correct process received the message through a R-Multicast protocol. In the beginning of a round,

Init:
1: $messagesToDeliver \leftarrow \emptyset$; $current\_agreement\_id \leftarrow 0$
2: $agreementStartRequired \leftarrow false$
3: **procedure** AB_Send(ID, message)
4: $R\_Broadcast(ID, message)$
5: **end procedure**
6: **procedure** $RB\_Deliver$(ID, message)
7: $messagesToDeliver \leftarrow messagesToDeliver \cup \{\langle ID, message\rangle\}$
8: **if** $agreementStartRequired$ **then**
9: REQUIRELOCALSTART($current\_agreement\_id$)
10: $agreementStartRequired \leftarrow false$
11: **end if**
12: **end procedure**
13: **procedure** CALCULATEPROPOSAL(id)
14: **if** $current\_agreement\_id < id$ **then**
15: **return** $\emptyset$
16: **end if**
17: $proposal \leftarrow \{\text{HASH}(m) : m \in messagesToDeliver\}$
18: **return** $proposal$
19: **end procedure**
20: **procedure** AGREEMENTFINISHED($id, result$)
21: **wait until** $(id = current\_agreement\_id) \wedge (\forall msg \in result :$ HASH$(msg) \in messagesToDeliver)$
22: $msgs \leftarrow \{msg \in messagesToDeliver :$ HASH$(msg) \in result\}$
23: **for all** $msg \in msgs$ **do**
24: $AB\_deliver(msg)$
25: **end for**
26: $messagesToDeliver \leftarrow messageToDeliver - result$
27: $current\_agreement\_id \leftarrow current\_agreement\_id + 1$
28: $agreementStartRequired \leftarrow true$
29: **if** $|messagesToDeliver| > 0$ **then**
30: REQUIRELOCALSTART($current\_agreement\_id$)
31: $agreementStartRequired \leftarrow false$
32: **end if**
33: **end procedure**

Fig. 6. Client/Initiator c_i of Atomic Broadcast

clients propose to the GCS the set of messages that must be delivered. Upon the arrival of the GCS consensus, the Consensus Filter Result function is applied to calculate an ordered list of stable messages to be delivered. Then, the clients will deliver this set of messages to application in the procedure `AgreementFinished()`. Only after the messages are delivered a new round is started.

1: **function** RESULT(vector)
2: $allMsgs \leftarrow \bigcup_{i=1}^{n_c} vector[i]$
3: $deliverableMsgs \leftarrow \{msg | \#_{msg} allMsgs \geq f_c + 1\}$
4: **return** DETERMINISTICORDER(deliverableMsgs)
5: **end function**

Fig. 7. Consensus Filter to solve Atomic Broadcast

VII. CONSENSUS PROTOCOL WITH TRUSTED COMPONENTS

The GCS provides separation between the processes responsible for executing the consensus protocol and the ones that want to solve an agreement problem. This separation of concerns allows us to use different fault models, or abstractions in servers and clients. We explore this property by using virtualization technology and trusted components to improve the servers' resilience to intrusion.

We propose two trusted components. Both components are used by processes in one VM to broadcast messages. They differ in the ordering constraints imposed on delivered messages. This components are implemented inside the VMM, so their protocols are subject only to crash faults.

The first component, called *postbox*, is the abstraction of an append-only shared memory region that allow efficient inter virtual machine communication. A process inside a VM can append a message to the postbox through a `Append(msg)` operation. Additionally, it may read messages from the postbox through a `Read()` operation that returns a message that was previously appended to the postbox. This component enforces that every process read the same set of messages in the same order. This constraint is satisfied by the underlying append only shared memory. Being all server processes on top of the same physical host machine, the service will not tolerate its crash.

The distribution of this component among several physical machine is computationally expensive, as it requires total ordering of the messages (i.e., a consensus protocol).

The postbox prevents equivocation of faulty servers, as well as enforcing a total order to the messages that are read from it. These guarantees allow the implementation of highly efficient consensus protocols. Under this model, consensus may be solved in one postbox communication step, and requires $n_s = 2f_s + 1$ processes to tolerate up to f_s faults.

The second component is called *distributed postbox*. It relaxes the ordering constraints of delivered messages, requiring that messages sent by a process are delivered in the same order in which they were sent. This change eases the distribution of the component among several physical machines.

In order to spread the *distributed postbox* among several physical machines, it is needed a reliable broadcast among VMMs in the crash fault model. Thus, the *distributed postbox* is preferable than the postbox if the crash of the physical machine is to be tolerated.

In the following, we present consensus algorithms based on the trusted components described previously. They are used by the servers of the GCS in order to achieve consensus. These algorithms are subject to Byzantine faults and may use the services provided by the trusted components in an arbitrary way.

A. Postbox-based Algorithm

The algorithm in figure 8 shows how the postbox abstraction can be used to implement a consensus protocols to be used among servers with 1 communication step that requires $n_s = 2f_s + 1$ servers to tolerate up to f_s faulty components.

The consensus algorithm operates by providing a procedure `Propose()`. When this procedure is called, the message is appended to the postbox. This algorithms requires a task to be running, responsible for monitoring the postbox. Whenever a proposal from a consensus instance is received, if it is the first one regarding that consensus instance then it is chosen as the consensus result.

Algorithm for process p_i:
1: **procedure** PROPOSE($id, proposal, certificate$)
2: APPEND($\langle$PROPOSAL$, id, proposal, certificate\rangle$)
3: **end procedure**
Task 1:
4: $decided_i^{id} \leftarrow false$
5: **loop**
6: $msg \leftarrow$ READ
7: **if** $msg = \langle$PROPOSAL$, id', proposal', certificate'\rangle$ **then**
8: $valid_i =$ CHECKCERT$(proposal', certificate')$
9: **if not** $decided_i^{id} \wedge valid_i$ **then**
10: $decided_i^{id} \leftarrow true$
11: DECIDE($proposal'$)
12: **end if**
13: **end if**
14: **end loop**

Fig. 8. Consensus algorithm using a postbox

The algorithm correctness is based on the fact that every server process receives the same set of messages in the same order from the postbox. Thus a process may decide the first valid message that arrives. If this process is correct, every other correct process behaves in exactly the same way.

`CheckCert()` procedure's code is omitted here. It is responsible for not allowing invalid proposal to be chosen as the consensus result.

B. Distributed Postbox-based Algorithm

The algorithm based on the distributed postbox algorithm shown in figure 9 is more complex due to weaker guarantees provided by the trusted component. Under the distributed postbox approach, each VMM provides access to a distributed postbox component. These components are interconnected through a private network avoiding attacks to the crash tolerant protocols.

Correctness proofs were developed, unfortunately due to space restrictions we will omit them. The proofs are available online at: http://www.inf.ufsc.br/~pieri/SGCProofs.pdf

The algorithm proposed here is round-based. Each round is asynchronous and has a leader. The leader of round r is the process with number $r \mod n$. In each round, the leader tries to impose its proposal to the other processes. The algorithm works as follows: when the round c starts its leader sends a message $\langle$PROPOSAL$; r_i; v_i^{id}; init_cert\rangle$ with the round number r_i, the leader's proposal v_i^{id} and a $init\_cert$ certificate. At the first round, $init\_cert = \emptyset$. When a process p receives the leader's proposal, it sends the message $\langle$PREPARE$; r_i; proposal\rangle$ where r_i is the round number and $proposal$ is the proposal received from the leader. If a process receives $f + 1$ prepare messages from the same round, it decides v and finishes. This behavior occurs when no faulty process is present and messages are received before any timer expires.

In order to guarantee liveness in the presence of faults a view change protocol may take place, causing the processes to advance to the next round. Whenever a process suspects that the leader is faulty, or it could not assemble $f+1$ prepare messages, it attempts to freeze the current round r_i, by sending a message $\langle$FREEZE$, r_i, v_p, r_p\rangle$. If the process sent a message

prepare during round r_i, $r_p = r_i$ and v_p is the proposal confirmed through the prepare message. Otherwise, v_p is a value proposed by the leader of round r_p, correct processes send v_p and r_p equals to the last freeze message that was sent by them. If no value was proposed yet, $v = \perp$ e $r_p = 0$.

The view change mechanism relies upon the construction of a valid $init\_cert$ (definition 7.1). A $init\_cert$ is a set of $f + 1$ valid freeze messages collected during round $r_i - 1$.

Definition 7.1: An $init\_cert$ is considered valid by a correct process if the conditions below are satisfied:

- Being r' the greatest r_p of freeze messages in $init\_cert$. Given any two messages freeze with $r_p = r'$ in $init\_cert$, namely m e m', then: $m.v = \perp$ or $m.v' = \perp$ or $m.v' = m.v$.
- All messages have correct signatures
- Given any freeze messages in $init\_cert$ in which $v \neq \perp$, the leader of round r_p proposed v.

Whenever a new leader proposes a value, it must certify its proposal with a valid $init\_cert$ certificate. An $init\_cert$ certifies values according to definition 7.2. The leader of the first round does not certify its proposal.

Definition 7.2: Given an $init\_cert$, and being r' the greatest r_p among freeze messages in $inic\_cert$. The value attested by $init\_cert$ is:

- A value $v \neq \perp$ of a message freeze with $r_p = r'$
- If all messages freeze in which $rp = r'$ have $v = \perp$, the any value is attested.

In order to simplify the specification of the algorithm, some verifications are isolated in a module, called *fault detection module*, responsible for detecting faulty processes analyzing the history of received messages. When a faulty process is detected all future messages from it are dropped, before the algorithm process them. After passing through this module, the following behavior is ensured:

- Round number of messages sent by one process are strictly increasing.
- If the primary proposes v in round r, then the primary must prepare v.
- If a process that has prepared v in round r freezes, if must freeze with value v and round r.
- If a process have not prepared in round r, it can only freeze a value w with round $r_p < r$. Being w the value proposed by the leader of round r.
- During a round, processes may send only one of these messages in the following order: proposal, prepare and freeze. The proposal must be made only by the round leader. A process may decide at any time.
- After freezing the round r, the next message a process send must regard the next round $r' = r + 1$.
- After the leader sends a propose, it must prepare. It can't freeze before preparing.
- All messages have valid certificates and signatures.

VIII. Comparisons and Results

First of all, it is important to notice that, in order to achieve generality the GCS sacrifices some possible performance optimizations. For instance, it requires that some messages must be digitally signed, which is widely known to be a common bottleneck in the system performance.

Init:
1: $r_i^{id} \leftarrow 0$; $init\_cert_i^{id} \leftarrow \langle \emptyset, \perp, \ldots, \perp \rangle$
2: $v_p \leftarrow \perp$; $r_p \leftarrow \perp$
3: **procedure** CONSENSUS($id, proposal, certificate$)
4: **loop**
5: **if** $round_i^{id}$ **mod** $n = i$ **then**
6: WRITEPOSTBOX($\langle$PROPOSAL, r_i, $v_i^{id}, init\_cert_i^{id}[r_i^{id}]\rangle$)
7: **end if**
8: **wait until** $proposals_i^{id}[r_i^{id}] \neq \perp$ **or** timed_out
9: $proposal\_received \leftarrow proposals_i^{id}[r_i^{id}] \neq \perp$
10: **if** $proposal\_received$ **then**
11: WRITEPOSTBOX($\langle$PREPARE, r_i, $proposals_i^{id}[r_i^{id}]\rangle$)
12: $r_p \leftarrow r_i$
13: $v_p \leftarrow proposals_i^{id}[r_i^{id}]$
14: **end if**
15: **wait until** timed_out
16: WRITEPOSTBOX($\langle$FREEZE, $r_i, v_p, r_p\rangle$)
17: **wait until** $init\_cert_i^{id}[r_i^{id}] \neq \perp$
18: $r_i^{id} \leftarrow r_i^{id} + 1$
19: **end loop**
20: **on receive** $\langle$PROPOSAL, $r_j, v_j, init\_cert_j^{id}\rangle$ **from** p_j
21: $proposals_i^{id}[r_j] \leftarrow v_j$
22: UPDATEINITCERT(id, r_j)
23: **on receive** $\langle$PREPARE, $r_j, proposal\rangle$ **from** p_j
24: $prepares_i^{id}[r_j] \leftarrow prepares_i^{id}[r_j] \cup \langle$PREPARE, $r_j, proposal\rangle$
25: **if** $\#_{proposal} prepares_i^{id}[r_j] = f + 1$ **then**
26: DECIDE($proposal.v$)
27: **end if**
28: UPDATEINITCERT(id, r_j)
29: **on receive** $\langle$FREEZE, $r_i, v_p, r_p\rangle$ **from** p_j
30: $freezes_i^{id}[r_j] \leftarrow freezes_i^{id}[r_j] \cup \langle$FREEZE, $r_i, v_p, r_p\rangle$
31: UPDATEINITCERT(id, r_j)
32: **end procedure**
33: **procedure** UPDATEINITCERT(id, r_j)
34: **if** $init\_cert_i^{id}[r_j] = \perp$ **then**
35: $x \leftarrow \{c \subseteq freezes_i^{id}[r_j] : |c| = f + 1 \wedge \forall e \in c : e.proposal = \perp \vee proposals_i^{id}[e.proposal.r] = e.proposal.v\}$
36: **if** $|x| > 0$ **then**
37: $init\_cert_i^{id}[r_j] \leftarrow$ **any element of** x
38: **end if**
39: **end if**
40: **end procedure**

Fig. 9. Distributed postbox-based consensus algorithm

Another point is the fact that some specific implementations may optimize common cases, leading to faster implementations in certain scenarios. For instance, the Byzantine consensus protocol in [20] solves agreement in one communication step under favorable conditions.

Assuming a GCS based consensus, we notice that GCS is more resilient to intrusion than classical consensus algorithms such as PBFT [3], requiring $2f + 1$ acceptors to execute the protocol. The latency required by GCS is two communication steps plus an access to the trusted component, instead of three required by PBFT.

In order to assess the performance of the proposed scheme, a prototype was implemented using Java 1.6. The test bed is composed by a Core 2 Quad CPU host machine, with 8 GB of RAM, running Debian GNU/Linux 5.0, kernel 2.6.26-2, and VirtualBox 2.2.4. Each virtual machines has 1 GB of RAM and access to one processor. Inside each VM, there is a

server running Ubuntu GNU/Linux 9.10, kernel 2.6.31-15. The clients run Ubuntu GNU/Linux 9.10 Desktop, kernel 2.6.31-15 in a Core 2 Duo CPU with 4 GB of RAM.

It was made micro-benchmarks in which each client proposed zero length messages to the consensus service. The consensus servers used the Postbox as the trusted component.

In the first experiment, we let the number of clients constant and increased the number of virtual machines. We observed that the throughput and response time remained constant, about 40 requests per second and 25ms respectively, until the limit of seven virtual machines were reached. At this point the throughput diminishes, while the response time increased. We believe that this happens because the computational power of the host machine starts to exhaust. There are eight cores, seven of them are processing the requests and the other one is managing the VMs and the postbox.

After that, we did the opposite: we let the number of servers constant while increasing the number of clients. In this scenario, the throughput decreases while the response time increases as the number of clients increases, as expected. The number of messages and signatures needed to be verified by the servers increases linearly with the number of clients, causing the the throughout to diminish linearly and the response time to increase linearly.

In order to assert the trusted component impact, we run one server ($f_s = 0$). In the first run we used the algorithm as is, using the postbox, however in the second run we changed the algorithm to not use the postbox. We observed a 10% decrease in throughput when using the postbox.

IX. Conclusions

This paper introduces the Generic Consensus Service, an extension of the previous consensus service towards the tolerance of Byzantine faults. We presented how the Generic Consensus Service works and how to specialize it to solve distinct agreement problems.

The separation of consensus protocol from agreement problems allowed the adoption of virtualization in part of the system, leading to the possibility of separating crash fault tolerance and malicious behavior treatment. We took advantage of this to implement a consensus agreement protocols that tolerates f malicious processes out of $2f + 1$ increasing the server's resilience to intrusion.

Besides that, we proposed two trusted components, and one consensus protocol for each trusted component. These trusted components increases the resilience of the system and allows to choose a tradeoff between crash fault tolerance and efficiency.

Finally, we commented the implementation of the GCS and compared some characteristics of a GCS based consensus algorithm to PBFT. Future works are focused on improving the initial prototype and the algorithms, solving other agreement problems with the GCS.

The main contributions of this paper are:

- Generalization of the consensus service presented by Guerraoui and Schiper to a Byzantine environment
- Definition of two trusted components implementable using virtualization technology
- Two Byzantine tolerant consensus protocols based on these trusted components
- Separation in tolerating crash and malicious faults through the use of the distributed postbox

References

[1] G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, New York, NY, USA, 1984. ACM.

[2] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 524–541, London, UK, 2001. Springer-Verlag.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. 571640.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. 226647.

[5] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. 2008. 1404038 287-292.

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their wordum. 2007. 1294280 189-204.

[7] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. 2002. 831132.

[8] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity byzantine-resilient consensus. *Distrib. Comput.*, 17(3):237–249, 2005. 1151559.

[9] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J.*, 49(1):82–96, 2006. 1183871.

[10] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. *Muteness Failure Detectors: Specification and Implementation*, pages 71–87. 1999.

[11] A. Doudou and A. Schiper. Muteness detectors for consensus with byzantine processes. 1998. 277772 315.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. 214121.

[13] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. Softw. Eng.*, 27(1):29–41, 2001. 359565.

[14] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[15] S. Kinney. *Trusted platform module basics: using TPM in embedded systems*. Newnes, 2006.

[16] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[17] H. P. Reiser and R. Kapitza. Vm-fit: Supporting intrusion tolerance with virtualisation technology. 2007.

[18] H. P. Reiser and R. Kapitza. Fault and intrusion tolerance on the basis of virtual machines. 2008.

[19] F. B. Schneider and L. Lamport. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, pages 431–480, London, UK, 1985. Springer-Verlag.

[20] Y. J. Song and R. Renesse. Bosco: One-step byzantine asynchronous consensus. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 438–450, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] A. Tomlinson. Introduction to the TPM. In *Smart Cards, Tokens, Security and Applications*, pages 155–172. 2008.

[22] P. Zieliński. Paxos at war. Technical report, 2004.