The Fail-Heterogeneous Architectural Model*

Marco Serafini and Neeraj Suri
Technical University of Darmstadt, Germany
{marco,suri}@cs.tu-darmstadt.de

Abstract

Fault tolerant distributed protocols typically utilize a homogeneous fault model, either fail-crash or fail-Byzantine, where all processors are assumed to fail in the same manner. In practice, due to complexity and evolvability reasons, only a subset of the nodes can actually be designed to have a restricted, fail-crash failure mode, provided that they are free of design faults. Based on this consideration, we propose a fail-heterogeneous architectural model *for distributed systems which considers two classes of nodes: (a) full-fledged* execution nodes*, which can be fail-Byzantine, and (b) lightweight, validated* coordination nodes*, which can only be fail-crash. To illustrate the model we introduce HeterTrust as a practical trustworthy service replication protocol. It has a low latency overhead, requires few execution nodes with diversified design, and prevents intruded servers from disclosing confidential data. We also discuss applications of the model to DoS attacks mitigation and to group membership.*

1. Introduction and motivation

Distributed services can be subject to malicious attacks which exploit vulnerabilities to take control of participant nodes and make them maliciously deviate from the specified operations. Similar to other design faults, security-related vulnerabilities could be eliminated by performing fault prevention and removal. However this is often impossible because fault removal processes (verification and validation) are time-consuming, do not scale well to complex systems, and can be ineffective in evolvable systems where new vulnerabilities can appear at any time, for example by installing faulty software or by incorrect configurations. As a consequence, intrusions and Byzantine failures in complex and evolvable systems needs to be considered and, if necessary, tolerated by means of replication and design-diversity.

The use of design-fault tolerance, achieved through design-diversity, as a complement to design-fault removal have been long debated in the safety-critical systems domain, where both options are viable thanks to small design sizes. However, the advantages of the first approach have not yet been substantiated by sufficient experimental evidence [19]. In fact, thorough fault removal, *if feasible*, can give high confidence in the absence of design faults, and thus justify the assumption of fail-crash behavior if random hardware faults are correctly handled (see e.g. [4, 25]).

Using fail-crash nodes with small designs can lead to significantly more efficient protocols without changing the overall distributed system model. Therefore, we introduce an *architectural* fault model based on a separation of concerns between two classes of nodes: full-fledged *execution nodes* containing the application logic and providing the service of interest, and lightweight *coordination nodes* providing only *restricted* coordination functionality. The model is *fail-heterogeneous* as no assumption is made on faulty execution nodes, while coordination nodes can only be fail-crash. It represents an intermediate step between the classic homogeneous fail-crash and fail-Byzantine models, and allows new tradeoffs between efficiency and safety.

Overall, we present the following contributions:

- We introduce and motivate the fail-heterogeneous architectural model, taking the problem of practical trustworthy state machine replication as a case study and presenting the HeterTrust protocol;
- We demonstrate that, by using a majority of coordination nodes with the same correct design, the minimal number of replicas with diversified design to tolerate f malicious faults can be reduced from $3f + 1$ [3] to $2f + 1$;
- We indicate how attackers can be prevented from disclosing confidential data of intruded servers by means of simple symmetric-key cryptography;
- We show that the latency overhead for replication and confidentiality with respect to a non replicated service is two communication steps;
- We discuss possible applications of the model to Denial-of-Service (DoS) attacks mitigation and to group membership.

*Research supported in part by DFG TUD-GK MM, EC DECOS and ReSIST

The paper is structured as follows: after discussing related work on state machine replication in Section 2, we introduce the HeterTrust protocol in Section 3 and prove its properties in Section 4. Further applications of the model are discussed in Section 5.

2. Related work

In order to substantiate the comparison with other models we consider different state machine replication approaches that are used to implement generic fault-tolerant services [26].

Byzantine agreement protocols and the homogeneous fail-Byzantine model were introduced to tolerate arbitrary physical faults in synchronous safety-critical systems [15]. Distributed systems designed to tolerate f Byzantine faults can in general handle $m \geq f$ less severe faults, although not necessarily at the same time. In order to model this, hybrid fault models [18, 22, 28] assume that *any* node in the system can fail in a malicious or benign manner, as long as upper bounds on the number of faulty nodes is kept.

Hybrid architectures partition the system into different subsystems with different sets of assumptions. For example, the Wormhole model [27] considers systems that are partitioned into multiple subsystems, which can be characterized by different failure modes and synchrony assumptions. An example of architecture under this model is TTCB [8], where each node is composed of two different subsystems. The first is an asynchronous, fail-Byzantine payload subsystem connected to the payload subsystems of the other nodes through an asynchronous payload channel. The second is a smaller synchronous, fail-crash control subsystem with limited computational capabilities and usually connected to the other control subsystems in other nodes through a dedicated, low bandwidth and synchronous control channel.

The fail-heterogeneous architectural model differs from hybrid fault models as it associates different fault models to specific nodes of the distributed system. It also differs from the Wormhole model as it does not consider different subsystems internal to nodes, nor different degrees of synchrony within specific subsystems.

The BFT protocol [5] for homogeneous fail-Byzantine systems implements state machine replication and guarantees that replicas do not diverge even in presence of malicious attacks and intruded participants. Compared to the Paxos protocol [16, 17], which is its fail-crash counterparts, BFT requires more replicas to tolerate f faults ($3f + 1$ instead of $2f + 1$) and has a higher latency. Subsequent work showed that a latency comparable to the crash-only case is achievable at the cost of a higher degree of replication ($5f + 1$) [21].

If agreement and execution are separated, as proposed in [29], agreement processes can have a simple design and require fewer local resources, while only $2f + 1$ complex replicas of the servers need to be diversified (using a proper abstraction layer such as [6]). However, to keep the number of faulty processes below the upper bound f there should be no correlation between failures, i.e., intrusions, at any different replica. As intrusions are made possible by design faults, i.e., vulnerabilities, failure independence requires diversified design of each *node* participating to the protocol (e.g., different operating systems must be used, different applications etc.) regardless of its role.

Based on similar considerations, our HeterTrust protocol assumes the availability of a set of simple nodes dedicated to replica coordination. The simplicity and generality of the consensus operation can justify a thorough verification and validation of the design of the nodes, which can then be reused in multiple contexts.

As pointed out in [29], replication of confidential data increases the likelihood that an attacker can intrude a replicated server and obtains confidential information. The authors of [29] propose a privacy firewall to make sure that (a) only replies processed by at least one correct process might be sent out by the service, and (b) replies should be as deterministic as possible to prevent attackers from using steganography. This represents the best solution proposed so far under a fail-Byzantine model. However, it requires a high number of replicas, a long latency to filter replies, and expensive threshold cryptography to make replies deterministic. The fail-heterogeneous architecture of HeterTrust represents a viable alternative to achieve the same properties with less overhead and fewer replicas.

HeterTrust uses a majority of correct fail-crash coordination nodes to reduce the number of complex fail-Byzantine execution nodes with diversified design to $2f + 1$. In [9], a similar result was achieved by relying on the synchrony of TTCB communication for agreement. HeterTrust tolerates periods of asynchrony and requires only an Ω leader election protocol (e.g. [1]) for liveness.

In [20] agreement and execution are also separated in fail-crash systems to take advantage of regions with "early" partial synchrony, where reaching agreement is easier. A hierarchical protocol decomposition approach for WANs is proposed in [2]. It allows choosing different combinations of fault tolerance protocols at each site and among sites in a customizable manner to mask Byzantine faults. Our approach differs from this as it binds failure modes to specific nodes based on their design.

Table 1 presents a comparison between HeterTrust and other deterministic state machine replication protocols. Most of the compared protocols assume partially synchronous system models similar to [11], except [9] where a Wormhole model is assumed. We report the upper bounds on crash (g) and Byzantine (f) faults tolerated. In general, only a subset of the nodes required for agreement (n) needs to actually implement the replicated service (e). Multi-tier

Table 1: HeterTrust - comparison with other deterministic state machine replication protocols

Protocol	SM	FM	*n*	*e*	*a*	Msg. complexity	Latency	Confid.	Crypt.
Paxos [16, 17]	PS	C	$2g+1$	$g+1$	-	$O(n^2)/O(n)$	$4/5$	-	-
BFT [5]	PS	B	$3f+1$	$3f+1$	-	$O(n^2)$	4	no	MAC
FaB [21]	PS	B	$5f+1$	$2f+1$	-	$O(n^2)$	3	no	MAC
Correia *et.al* [9]	W	W	$2m+1$	$2m+1$	-	$O(n^2)$	5	no	MAC
Marchetti *et.al* [20]	PS	C	$e+a$	$g+1$	$2g+1$	$O(n^2)/O(n)$	$4/5$	-	-
Yin *et.al* [29]	PS	B	$e+a$	$2f+1$	$f+1/f^2+4f+1$	$O((3f+1)^2)$	$4/2f+7$	no/yes	MAC/TS
HeterTrust	PS	H	$e+a$	$2f+1$	$2g+1$	$O(a^2+a\cdot e)$	4	yes	MAC

$n/e/a$ = lower bound on # nodes / execution processes / additional nodes; $g/f/m$ = upper bound on # fail-crash / Byzantine / mixed nodes;
SM = System Model (**P**artially **S**ynchronous / **W**ormhole); **FM** = Fault Model (**C**rash / **B**yzantine / **W**ormhole / **H**eterogeneous)
MAC = Message Authentication Codes; TS = Threshold Signatures

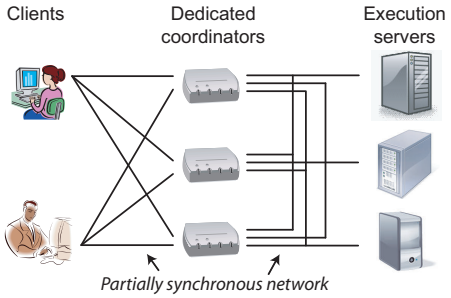

Figure 1: A fail-heterogeneous architecture

architectures require additional nodes (a), for confidentiality or for faster agreement. In this case g and f represent upper bounds for each layer. The state machine message complexity and latency is measured during best-case runs as the number of communication steps on the critical path from a client request to its reply. Where indicated by the authors, we consider the use of tentative executions [14]. For confidentiality, additional communication steps are necessary in [29] and HeterTrust. A simple variation of HeterTrust, which does not provide confidentiality and allows saving one communication step, is discussed briefly in Section 3.3. Table 1 also indicates the type of cryptography used during normal operation in the critical path. All mentioned fail-Byzantine protocols use public keys during recovery.

3. The HeterTrust protocol

In this section we describe HeterTrust as a practical trustworthy state machine replication protocol for partially synchronous systems. It uses dedicated coordination nodes (called coordinators in the following) to order client requests and filter replies coming from the execution nodes (called servers in the following) for confidentiality. Coordinators have use the same design, whereas servers use a diversified design.

3.1. System model

The system is composed by c fail-crash coordinators, s fail-Byzantine servers and a bounded number of authenticated clients. If the system is to tolerate up to g coordinator crashes and up to f Byzantine servers, we assume to have $c \geq 2g+1$ coordinators and $s \geq 2f+1$ servers. The protocol tolerates any number of malicious clients. Participants communicate through an asynchronous, unreliable network. Point-to-point authenticated channels among participants are ensured by means of MACs, i.e., symmetric cryptography, and collision-resistant cryptographic hash functions (to produce digests). For liveness, we additionally assume that channels between any pair of correct hosts are fair-lossy, i.e., they eventually deliver messages that are repeatedly resent, and that coordinators execute a leader election protocol which eventually elects a single leader (commonly called Ω). This becomes possible as soon as the system has enough timely links [1]. In order to provide confidentiality, coordinators must be physically interposed between clients and servers (see Fig. 1).

3.2. Service properties

The protocol allows clients to send requests to the trustworthy replication service through the coordinators. The leader among the coordinators assigns a progressive sequence number to each received request and sends it to the servers, which execute the request and send it back, together with the reply, to all coordinators. These filter out spurious or incorrect replies and forward the correct ones to the client, which delivers it (see Fig. 2). Formally, the properties provided by the trustworthy state machine replication service are the following (adapted from [20]):

Termination: If a correct client cl issues a request req then it eventually delivers a reply $repl$.

Uniform Agreed Order: If a correct server computes a request req as the i^{th} request, then every correct server that processes the i^{th} request must process req as the i^{th} request.

Update Integrity: For each request req, every correct server computes the request at most once, and only if a client has issued req.

Response Integrity: A client receives a reply $repl$ only if it has sent a request req and at least one correct server has sent $repl$. If the client is correct, at least one correct server has computed $repl$ as a reply to req.

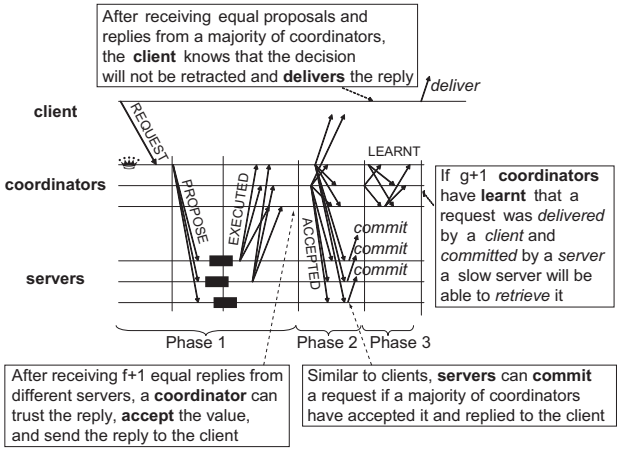

Figure 2: HeterTrust: Normal operations

Termination is the liveness condition of the service. *Uniform Agreed Order* prevents correct servers from diverging and is sufficient for linearizability, i.e., clients sending concurrent requests to the service can observe the same course of action. *Update Integrity* guarantees the "exactly-once" semantic. *Response Integrity* enforces both integrity and confidentiality as it requires filtering out spurious and incorrect replies from Byzantine servers. Additionally, HeterTrust ensures that a faulty server cannot use nondeterminism in content of the reply message to convey hidden confidential information via steganography.

The algorithms executed by the clients, by the coordinators during normal operations and during recovery, and by servers are Alg. 1, 2, 3 and 4 respectively. Table 2 explains the local variables used by the processes and their initial values. In the following, we describe (a) the normal operations of the protocol and (b) the recovery from leader crashes.

3.3. Normal operations

We first describe runs where there is single correct leader coordinator endorsed by all correct coordinators and there is no message loss. In these runs, the protocol proceeds through three phases upon the reception of a request from a client (see Fig. 2). In Phase 1, it tries to provide a quick answer to the client. In Phase 2, it goes through an additional coordination step to let coordinators and servers know about the reply (potentially) delivered by the client. Finally, in Phase 3 it ensures that slow servers can directly retrieve old requests from at least one coordinator without triggering other instances of the agreement protocol.

Phase 1: Replying to clients. When the client wants the service to perform an operation *op*, it assigns it a locally unique timestamp t and forms a request req, which is enqueued in $requests$ (line 4). Correct clients send only one request at a time, and locally queue requests if they are already waiting to deliver the reply for another request. The task *sendRequests* periodically sends requests to the coordinators (lines 14–18). It then initiates the protocol by sending a REQUEST message to all coordinators until they can deliver a reply (lines 1–4).

Table 2: Local Variables (for sequence no. i)

Name	Description	Initial value
Acc	set of accepted and non retrievable requests	-
$acc[i, co]$	last ACCEPTED message received from coordinator co	$\perp$
$Accepted$	set of IDs of the coord. which sent equal ACCEPTED messages	-
$accval[i]$	accepted request	$\perp$
$bComm$	buffer of requests to be committed	$\emptyset$
$bProp$	buffer of requests to be executed	$\emptyset$
$commReply[cl]$	proposal and reply of the last committed request from client cl	$(\perp, \perp)$
$endGap$	end of a gap in the requests received by a server	-
$endorse$	proposal number of the last observed leader	0
$exec[i, se]$	last EXECUTED message received from server se	$\perp$
$Executed$	set of IDs of the coord. which sent equal ACCEPTED messages	-
$lastComm$	sequence number of the last committed request	0
$lastDel$	timestamp of the last delivered request	0
$lastProp$	proposal number of the last executed request	0
$Learnt[i]$	set of IDs of the coordinators which sent a LEARNT message	$\emptyset$
$learntval[i]$	learnt request	$\perp$
$maxAcc$	max seq. number with accepted but not retrievable req. (as from ENDORSE msgs)	-
$maxAcc$	max seq. number with all previous retrievable req. (as from ENDORSE msgs)	-
$Ongoing$	set of IDs of the client with not retrievable requests	$\emptyset$
op	requested operation	-
$Proposals$	set of accepted requests (as from ENDORSE msgs)	-
$repl$	reply to requests	-
$requests$	local queue of requests of a client	$\emptyset$
$Retr$	set of sequence numbers bound to a retrievable request	$\emptyset$
t	timestamp to identify requests from a client	0
tEx	current tentatively executed request and related reply	$(\perp, \perp)$

When the leader coordinator receives a request (line 20), it forms a *proposal* $(req, prop)$ attaching a proposal number $prop$ to the request, which is used by the other coordinators to discard messages coming from old leaders. Each coordinator is assigned a partition of the set of positive integers. Upon election, a leader coordinator increases its proposal number until it becomes the highest observed by enough others participants, which will then *endorse* it (see Section 3.4 for details). A leader proposes only a bounded number of requests in parallel and queues the remaining requests (at most one for each client, line 21).

The proposal is then given an increasing sequence number i, stored in $propval[i]$, and sent in a PROPOSE message to all servers by the task *sendPropose* (lines 54–58). Following the terminology of [17] the request is now *proposed*. The sequence number will be used by each correct server to order the execution of requests and thus to keep a

Algorithm 1: Client cl

```
1  upon initiate(op)
2      t := t + 1;
3      req := (op, t, cl);
4      enqueue(requests, req);
5
6  upon accepted(i, req, prop, repl) from coordinator co
7      if lastDel ≥ req.t then
8          acc[i, co] := (req, prop, repl);
9          Accepted := {co | acc[i, co] = (req, prop, repl)};
10         if |Accepted| ≥ ⌈(c + 1)/2⌉ then
11             deliver(req.op, req.t, repl);
12             lastDel := req.t;
13
14 task sendRequest
15     while (req := dequeue(requests)) ≠ ⊥ do
16         repeat
17             send (REQUEST, req) to all coordinators; wait timeout;
18         until lastDel < req.t ;
19
```

Algorithm 2: Coordinator - normal operations

```
20 upon request(req)
21     if leader() ∧(req.cl ∉ Ongoing) then
22         Ongoing := Ongoing ∪ {req.cl};
23         i := nextSequenceNumb() ;
24         propval[i] := req;
25
26 upon executed(i, req, prop, repl) from server se
27     if prop ≥ endorse then
28         exec[i, se] := (req, prop, repl);
29         Executed := {se | exec[i, se] = (req, prop, repl)};
30         if |Executed| ≥ f + 1 then
31             accval[i] := (req, prop);
32             send (ACCEPTED, i,req,prop,repl) to client req.cl;
33             send (ACCEPTED, i,req,prop) to all coord. and servers;
34
35 upon accepted(i, req, prop) from coordinator co
36     acc[i, co] := (req, prop);
37     Accepted := {co | acc[i, co] = (req, prop)};
38     if |Accepted| ≥ ⌈(c + 1)/2⌉ then
39         learntval[i] := (req, prop);
40         send (LEARNT, i, req, prop) to all coordinators;
41
42 upon learnt(i, req, prop) from coordinator co
43     if learntval[i] = ⊥ then
44         learntval[i] := (i, req, prop);
45         send (LEARNT, i, req, prop) to all coordinators;
46     Learnt[i] = Learnt[i] ∪ {co};
47     if |Learnt[i]| ≥ g + 1 then
48         Retr := Retr ∪ {i};
49         Ongoing := Ongoing \ {req.cl};
50
51 upon retrieve(i) from server se
52     if (learntval[i] ≠ ⊥) then send (LEARNT, learntval[i]) to se ;
53
54 task sendPropose
55     while leader() do
56         foreach i ∉ Retr do
57             send (PROPOSE, i, propval[i], prop) to all servers;
58         wait timeout;
59
```

consistent state with the other correct servers. As long as there is only one leader coordinator, a single request will be assigned a unique increasing sequence number.

On receiving a PROPOSE message from the current leader, the servers produce a reply $repl$ (lines 86–96). Similar to [5, 14], new requests are only *tentatively* executed, i.e., the changes to the service state are written in a temporary log before being *committed*. If the leader crashes the new leader can change the order of some requests, and this can cause tentative executions to *roll back*. Otherwise, tentative executions are eventually and definitively committed.

Servers should only accept messages from the latest leader. For this purpose, they store the highest proposal number they have observed ($lastProp$). They also store the sequence number of the last committed request ($lastCommit$) and only execute the next request (with sequence number $lastCommit + 1$). Requests with higher sequence number are buffered in $bProp$ unless they have been already buffered or if they come from a previous leader (lines 97–99).

When servers process a request (line 89) they check if this has already been committed (line 90) or tentatively executed (line 92), and retrieve the previous reply in these cases. If not, then they obtain the reply by tentatively executing the request possibly after performing a rollback of previous tentative executions (lines 93–95). Servers attach the reply, together with the proposal, in an EXECUTED message sent to all the coordinators.

The coordinators ignore proposals from previous leaders (line 27). They also filter out malicious and spurious replies from servers by waiting for $f + 1$ equal EXECUTED messages (lines 28–30). This ensures that the reply was sent by at least one correct server and that it is an actual reply to a request proposed by the leader. In this case coordinators *accept* [17] the proposal by storing it in the variable $accval[i]$ (line 31) . It then notifies, through an ACCEPTED message, all coordinators, servers and the client $req.cl$ which issued the request (lines 32–33). The ACCEPTED message sent to the client also contains the correct reply.

When the client receives an ACCEPTED message (line 6) for an ongoing request (line 7), it knows that the reply to its request was tentatively executed by at least one correct server. However, such a reply will only be delivered after it is guaranteed that this tentative execution will not roll back. As discussed in Section 3.4, the recovery protocol ensures that if a request is *chosen* for a sequence number [17], i.e., it is contained in a proposal that is accepted by a majority of coordinators, then its execution will never be rolled back even if the leader and other coordinators crash. The client thus waits until it receives ACCEPTED messages for the same proposal from a majority of coordinators before delivering the reply (line 11). Thus, after four communication steps a client can deliver the reply.

If confidentiality is not required, one communication step can be saved by having servers send EXECUTED messages directly to the clients, which can thus filter out incorrect replies by waiting for $f + 1$ equal replies. In this case, the leader sends PROPOSE messages to servers and coordinators in the same communication step, and clients will deliver a correct reply only after receiving ACCEPT mes-

Algorithm 3: Coordinator - recovery

```
60 upon elected()
61     repeat
62         prop := nextPropNumb();
63         send (QUERY, prop, Retr) to all coordinators;
64         wait timeout;
65     until ( receive ( ENDORSE, prop, Acc_co, Retr_co ) from
       ⌈(c + 1)/2⌉ coord co ) or ( ¬ leader () ) ;
66     if ¬ leader () then return;
67     Retr := Retr ∪_co Retr_co;
68     maxRetr := max{i | (i ∈ Retr) ∧ (∀ j ≤ i, j ∈ Retr)};
69     maxAcc := max{j | ∃accval, co : (j, accval) ∈ Acc_co};
70     propval := ⊥;
71     foreach j ∈ [max(Retr) + 1, maxAcc] : j ∉ Retr do
72         Proposals := {accval | ∃co : (j, accval) ∈ Acc_co};
73         if Proposals ≠ ∅ then
74             propval[j] := req ∈
               Proposals with max{req.prop};
75         else propval[j] := no_op;
76
77 upon query(prop, Retr_l) from coordinator l
78     if prop > endorse then
79         endorse := prop;
80         Retr := Retr ∪ Retr_l;
81         Acc := ∅;
82         foreach j : (j ∉ Retr) ∧ (accval[j] ≠ ⊥) do
83             Acc := Acc ∪ {(j, accval[j])};
84         send (ENDORSE, endorse, Acc, Retr);
85
```

sages by a majority of coordinators.

Phase 2: Committing the reply. In order to ensure progress, coordinators take additional steps to guarantee that the servers can commit tentative executions. Coordinators and servers try to determine if a request was chosen for a sequence number and is therefore indissolubly bound to it. Similar to clients, they do this by waiting for ACCEPTED messages by a majority of coordinators (lines 35–40 and 101–104). When this happens, the request is *learnt* for a sequence number [17]. Coordinators store learnt requests for sequence number i in the variable $learntval[i]$ (line 39), and communicate this to all the other coordinators (line 40). Coordinators can also learn that a request was chosen by receiving a LEARNT message (line 42–45).

A server learns that a request was chosen (line 106) if it has sequence number $lastCommit + 1$. Commits for higher sequence numbers, as well as requests, are buffered in $bComm$ (line 107). If a chosen request has not already been committed (line 109) it is tentatively rollbacked and executed as necessary and then committed (lines 110–114). Subsequently, further buffered requests for the next sequence numbers, which have been learnt or proposed, can be processed (lines 116–120).

Phase 3: Handling slow servers and message losses. Some servers might not learn that a request was chosen, either because they are slow or due to message losses. This prevents them from committing a tentative execution, and thus from executing further requests they receive. In this case the task *fillGaps* sends a RETRIEVE message to the coordinators to learn the chosen request (lines 122–126 and 51–52). To guarantee that at least one coordinator will

Algorithm 4: Server

```
86 upon propose(i, req, prop) from coordinator
87     if prop ≥ lastProp then
88         lastProp := prop;
89         if i = (lastComm + 1) then
90             if commReply[req.cl].t ≥ req.t then
91                 repl := commReply[req.cl].repl;
92             else if tEx.req = req then repl := tEx.repl;
93             else
94                 if tEx.req ≠ ⊥ then rollback(tEx.req);
95                 repl :=execute(req); tEx := (req, repl);
96             send (EXECUTED, i, req, prop, repl) to all coordinators;
97         if i > (lastComm + 1) then
98             if prop > bProp[i].prop then
99                 bProp[i] := (req, prop);
100
101 upon accepted(i, req, prop) from coordinator co
102     acc[i, co] := (req, prop);
103     Accepted := {co | acc[i, co] = (req, prop)};
104     if |Accepted| ≥ ⌈(c + 1)/2⌉ then learnt(i, req, prop);
105
106 upon learnt(i, req, prop) from coordinator
107     if i > (lastComm + 1) then bComm[i] := (req, prop);
108     if i = (lastComm + 1) then
109         if commReply[req.cl].t < req.t then
110             if tEx.req ≠ req then
111                 if tEx.req ≠ ⊥ then rollback(tEx.req);
112                 repl := execute(req);
113             commReply[req.cl] := (req.t, repl);
114             commit (req, i);
115         tEx := (⊥, ⊥); i := i + 1;
116         lastComm := lastComm + 1;
117         if bComm[i] ≠ ⊥ then
118             learnt(i, bComm[i].req, bComm[i].prop);
119         else if bProp[i] ≠ ⊥ then
120             propose(i, bProp[i].req, bProp[i].prop);
121
122 task fillGaps
123     while true do
124         wait timeout; endGap := min{j | (j >
           lastComm) ∧ ((bProp[j] ≠ ⊥) ∨ (bComm[j] ≠ ⊥))} ;
125         foreach j ∈ [lastComm + 1, endGap − 1] do
126             send (RETRIEVE, j) to all coordinators ;
127
```

be able to reply to RETRIEVE messages, the leader has to keep sending PROPOSE messages and thus push protocol messages until it receives $g+1$ LEARN messages from different coordinators (lines 46–48 and 56–58). This enables correct servers to recover from message losses, but prevents malicious servers from flooding the system by triggering consensus instances for request retrieval. A request is thus called *retrievable* if at least $g+1$ coordinators have learnt it. After a leader knows that a request from a client is retrievable, it can also accept further requests from *cl* (lines 49 and 21).

3.4. Recovery

Due to system asynchrony and crashes, the leader election protocol can output multiple coordinators as leaders, possibly at the same time. It is then necessary to prevent newly elected leaders from retracting decisions which already caused irreversible evolutions of the system state, such as the delivery of a reply to a client or the commit of an execution done by a server. In particular, if a request is chosen, i.e., it is accepted by a majority of coordinators, it is

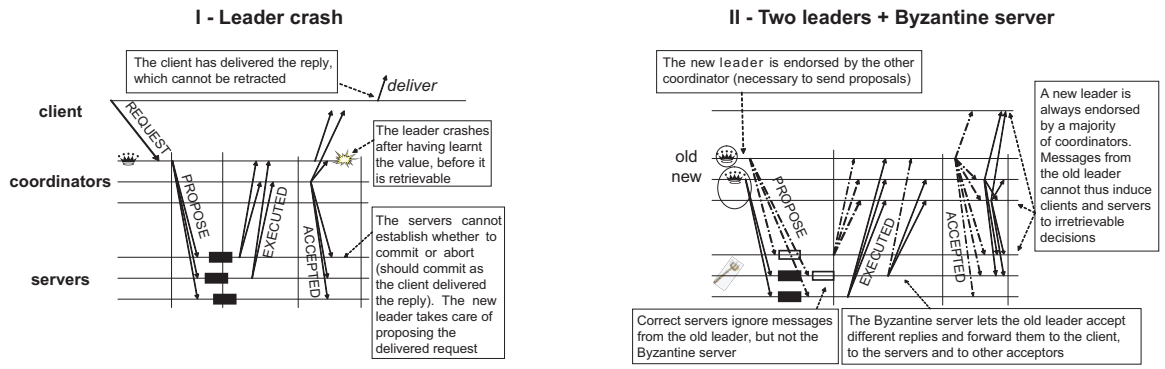

Figure 3: Two fail-prone scenarios

necessary to prevent new leaders from proposing different requests and having them accepted. Consider for example scenario (I) of Fig. 3. In this case, the new leaders must ensure that servers will commit the (chosen) request used to compute the reply delivered by the client. To guarantee this, the protocol adopts a recovery procedure (Alg. 3) which is similar to the one used by the Paxos protocol [16]. The similarity is given by the fact that only fail-crash participants (i.e. the coordinators) are involved.

Upon being elected, a leader selects a new proposal number and sends a QUERY message asking all other coordinators to endorse it (lines 60–65). It also asks them if (a) they have accepted some request $accval[j]$ for the sequence numbers that are not yet bound to a retrievable request in its local view (i.e., they are not in $Retr$), or (b) they know that there is a retrievable request for these numbers. Unless the other coordinators have already endorsed another leader with higher sequence number (line 78), they endorse the new leader (line 79) and form their set of accepted requests Acc and retrievable requests $Retr$ (which do not require further operations). They then send both sets to the new leader (lines 80–84).

Upon receiving ENDORSE messages from a majority of coordinators, the leader can start proposing requests (lines 67–75. For sequence numbers with an associated retrievable request, no operation is needed. For other sequence numbers where some accepted request is reported, the new leader must send its proposals without contradicting previously chosen requests. This is done by selecting the proposal from the latest previous leader, i.e., the one with the highest proposal number (lines 74). Gaps are filled with special no_op requests (line 75). If a request proposed from a certain leader is accepted by a majority of coordinators, each subsequent leader will receive notification of it in at least one ENDORSE message and select it for proposal. This guarantees that chosen requests are not overwritten.

It is surprising how efficiently a Paxos-like recovery protocol under the fail-heterogeneous model tolerates the effect of Byzantine faults at the servers, even using tentative executions. For example, in scenario (II) of Fig. 3, two leaders are simultaneously present and a Byzantine server sends them inconsistent information. Each leader waits for an endorsement from a majority of coordinators before issuing proposals, and Byzantine servers are not involved in this decision. Although servers can forward messages from an old leader to a minority of coordinators and have them accepted, these coordinators cannot induce correct clients and servers to take wrong delivery or commit actions.

3.5. Garbage collection

Servers can discard all data regarding sequence numbers of committed requests. Coordinators could as well garbage-collect the data structures of sequence numbers of retrievable request, but they need to indefinitely keep them in $learntval[i]$ to reply to RETRIEVE messages from slow servers. To avoid this, a simple checkpointing protocol is used. This is not included in the previous algorithms, but we describe it briefly.

When the commit procedure is required to commit a request with sequence number i such that $(i \mod k) = 0$ for a given checkpoint frequency k, it produces a tentative checkpoint of local service state, calculates a digest of it and sends a (CHECKPOINT, i, cpd) message with the MAC of the digest cpd to all coordinators. After receiving $f+1$ such equal messages from different servers, coordinators know that at least one correct server has an available checkpoint of the service state up to sequence number i. They then send an (ACKCP, i) message to all servers. When a server receives $g + 1$ such messages, it can delete the previous checkpoints, complete the commit procedure, and start processing the next executable requests.

A coordinator receiving a checkpoint digest from $f + 1$ servers for sequence number i knows that it can garbage collect entries previous to $learntval[i]$ as, if necessary, slow servers trying to retrieve old chosen requests can be sent a complete checkpoint. However, to minimize state transfers,

it tries to reply with simple requests when possible. Therefore, it only deletes entries of learnt requests in the array $learntval$ for sequence numbers preceding the prior checkpoint, i.e., prior to $i - k + 1$. A slow server receives the checkpoint state for sequence number i only if it tries to retrieve a request for a sequence number $j \leq i - k$. In this case, coordinators obtain the checkpoint state by sending a (QUERYCP, i) message to all servers until they receive at least one correct checkpoint, which is recognized using the digest. They then store the checkpoint state (at most one at a time) for further requests, and send it to the slow server.

4. Proof of correctness

In this section, we prove that HeterTrust satisfies the specified properties of a trustworthy replicated service. As mentioned in Section 3, we follow the terminology of [17]. A *request* req is *proposed* if it is issued by a leader with proposal number $prop$. A *proposal* is the tuple $(req, prop)$. A proposal, and therefore the associated request and the corresponding reply, is *accepted* if one coordinator accepts it. This happens if the proposal comes from a leader that the coordinator currently endorses, or a following one with a higher proposal number. As coordinators receive proposals through $f + 1$ servers, Lemmas 1 and 2 guarantee that accepted proposals have been sent by a leader coordinator and replied by at least one correct server. If the request is issued by a correct client, it is accepted together with the corresponding correct reply (Lemma 3). A proposal (and the relative request) is *chosen* if a majority of coordinators accepted it. Only a single chosen proposal is indissolubly bound to a sequence number (Lemma 4). Based on this property, clients and servers can take irreversible actions on requests (i.e., deliver them and commit them) if they receive a majority of ACCEPTED messages and thus learn that the request was chosen (Lemma 5). A request is *retrievable* if it is chosen for a sequence number i and $g + 1$ coordinators have learnt it. As leaders continue sending requests for a sequence number until they become retrievable, eventual progress is guaranteed even if correct servers are temporarily disconnected and do not commit old requests (Lemma 6). Finally, we prove the required properties of the protocol in Theorem 1.

Lemma 1 *Only a request req that has been proposed by a coordinator is accepted for a sequence number i, together with a reply $repl$ obtained from at least one correct server.*

Proof By definition, a request is accepted for a sequence number i only if it is contained in a proposal $(req, prop)$ that is accepted by any coordinator (line 31). A coordinator accepts a proposal $(req, prop)$ for i only after it receives $f + 1$ equal (EXECUTED,$req, i, prop, repl$) messages from different servers (lines 28–30). Among these servers, at least one must be correct. This has thus sent the EXECUTED message (line 96) containing values i, req and $prop$ as from the message proposed by a leader coordinator and the reply $repl$.

Lemma 2 *Only a request req that has been proposed by a coordinator is chosen for a sequence number i, together with a reply $repl$ obtained from at least one correct server.*

Proof This follows directly from Lemma 1 as a chosen request must be accepted.

Lemma 3 *If a request req is issued by a correct client and accepted, it is accepted together with a reply $repl$ which has been computed by at least one correct server calling* execute*(req).*

Proof If a request req is accepted together with a reply $repl$, at least one correct server has sent a message (EXECUTED,$req, i, prop, repl$) (line 96) after receiving a proposal from a coordinator (Lemma 1). If the request is issued by a correct client $req.cl$, which sends only one request at a time with growing timestamp $req.t$ (lines 1–4), no correct server can have seen a request from the same client with higher timestamp. Therefore, no correct server has stored a reply such that $commReply[req.cl].t \geq req.t$ (line 90). The reply is thus obtained from a tentative execution of the same request possibly having rolled back any previous tentative execution (lines 92–95).

Lemma 4 *Only a single request is chosen for a sequence number i.*

Proof A request req is chosen for i when a proposal $(req, prop)$ is chosen, i.e., accepted by a majority of coordinators. By definition, only one proposal can be chosen at a time. Assume that $p_1 = (req_1, prop_1)$ is the first proposal chosen for i and that it is proposed by the leader l_1. In order for any another proposal $p_2 = (req_2, prop_2)$ with $req_1 \neq req_2$ to be chosen, it is necessary that at least one of the coordinators that accepted p_1 accepts p_2 afterwards.

From Lemma 1, any accepted value has been proposed by a leader. As leaders never change their proposals until demoted and re-elected, p_2 must have been issued with proposal number $prop_2 \neq prop_1$. Therefore, a coordinator accepts the new proposal p_2 after having accepted $p_1 = accval[i]$ only if p_2 has a higher proposal number $prop_2 > prop_1$ (line 27). We show that any chosen proposal p_2 issued after p_1 is such that $req_2 = req_1$. Let us assume, by contradiction, that p_2 is the issued proposal with the minimum proposal number $prop_2 > prop_1$ such that $req_2 \neq req_1$.

When l_2 is elected, it sends a QUERY message to all coordinators and sends new proposals only after it receives

ENDORSE messages from a majority of them (lines 60–65). At least one of the coordinators member of the majority which accepted $p_1 = accval[i]$ must have sent an ENDORSE message reporting either that (a) req_1 is retrievable ($i \in Retr_{co}$) or (b) p_1 was accepted ($p_1 \in Acc_{co}$) (lines 80–83). In the first case l_2 does not send any new proposal for i (line 71). Therefore, if l_2 proposes $proposal = (req_2, i, prop_2)$ with $req_2 \neq req_1$, there must exist a coordinator which reports to have accepted req_2 from a leader $l_3 \neq l_2$ with a proposal number $prop_3 > prop_1$ (lines 71–74). To accept the proposal of l_3 and to report it through an ENDORSE message to l_2, this coordinator must have endorsed first l_3 then l_2 (lines 79 and 27), and this implies $prop_2 > prop_3$. Therefore, p_2 is not the accepted proposal with the minimum proposal number greater than $prop_1$, which is a contradiction.

Lemma 5 *Only a reply to a chosen request for sequence number i can be delivered by a client, and only a chosen request for sequence number i can be learnt by a coordinator or committed by a server.*

Proof Coordinators send an ACCEPTED message containing a proposal only after accepting it (lines 31–33). Receiving ACCEPTED messages from a majority of coordinators is a necessary condition for clients to deliver a reply (line 10). Coordinators and servers learn that a request is chosen either by the same condition (lines 38 and 104), or by receiving a LEARNT message (lines 42 and 106), which is sent only after some coordinator has learnt that the request was chosen (line 40). When a server learns that a request was chosen, it commits it, executing it unless it has already been tentatively executed (lines 104 and 109–114).

Lemma 6 *Some proposed request for a sequence number i is eventually chosen and becomes retrievable.*

Proof The proof is by induction on the sequence numbers, assuming that a no_op request with sequence number 0 is trivially chosen and retrievable. Assume that requests for sequence numbers $j < i$ have been chosen and are retrievable. From the property of the leader election protocol, eventually a single correct leader is elected. By definition, only chosen proposals can be retrievable. If a proposal for i is already chosen but not retrievable, the leader proposes it to the servers (Lemma 4) until it becomes retrievable (lines 54–58). If no request for i is chosen but the leader has received a request to serve, such request is associated to i and proposed to the servers (lines 20–24). As all requests with sequence numbers $j < i$ are retrievable, correct servers can eventually obtain them from at least one correct coordinator (lines 122–126 and 51–52) and commit them (lines 106 and 109–114). After that the $s - f \geq f + 1$ correct servers can process request i (lines 89–95) and send the corresponding EXECUTED message to the coordinators (line 96), which then accept the proposal (line 31). The $c - g \geq \lceil (c+1)/2 \rceil$ correct coordinators forward ACCEPT messages to each other, until eventually all of them will learn the request (line 38–39) and make it eventually retrievable, by exchanging LEARNT messages (lines 40, 42 and 46).

Theorem 1 *The HeterTrust protocol satisfies the properties of Termination, Uniform Agreed Order, Update Integrity and Response Integrity.*

Proof The service properties specify how servers should *compute* requests. These requirements refer, in the context of this protocol, to committed executions of requests. In the following we prove that each required property is met.

Termination: By repeatedly sending its request (line 14–18), a client can ensure that each request is eventually received by all coordinators. A correct client sends a request only after previous requests are delivered. Therefore, a leader will eventually propose it. From Lemma 6, some proposed request is eventually chosen for each sequence number, and becomes retrievable. If there are no message losses, the client receives ACCEPT messages from a majority of coordinators and delivers a reply. In case of message losses the client re-issues the same request until it eventually receives enough ACCEPT messages (possibly with another sequence number).

Uniform Agreed Order: A correct server commits only chosen requests (Lemma 5). If the request req is committed, and thus chosen, as the i^{th} request, Lemma 4 ensures that any other correct server that commits the i^{th} request will commit req.

Update Integrity: Each request req is uniquely identified by its sender $req.cl$ and its timestamp $req.t$. Servers keep an array $commReply$ with the timestamps of the last committed requests replied to each client, together with the corresponding reply (line 113). A request req from a client cl is executed only if it has a higher timestamp than $lastReply[cl]$, else the cached reply is resent (lines 93–90). Therefore, a committed request is never executed again. Furthermore, each committed request req with $req.op \neq no\_op$ is issued by a client. In fact, only chosen values are committed (Lemma 5), only proposed values are chosen (Lemma 2) and a request req with $req.op \neq no\_op$ is proposed by a leader coordinator only if it is received from a client (line 54).

Response Integrity: As coordinators are physically interposed between servers and clients, clients can receive replies rep (as well as any other data) from servers only through ACCEPTED messages sent by coordinators. These are sent only for accepted requests, which are obtained by at least one correct server (Lemma 2). If the client is correct,

from Lemma 3 it follows that the reply $repl$ is obtained by the correct server as *execute*(req).

5. Further applications of the proposed model

HeterTrust represents only one example of possible problems which can be efficiently solved using the fail-heterogeneous architectural model. In this section, we discuss two other possible applications of the model, namely, liveness under DoS attacks and group membership.

5.1. DoS-attacks and consensus liveness

HeterTrust is safe in periods of asynchrony but relies on additional synchrony for liveness [12]. If we can expect that these properties are generally met by the network in a benign (i.e., crash-only) environment, they cannot be in general preserved if an attacker is able to introduce "malicious asynchrony" and make communication unreliable by means of Denial of Service (DoS) attacks. Therefore, unless specific countermeasures are taken, trustworthy service replication systems can easily be made unavailable by simple DoS attacks. Dedicated coordination nodes can be used as active participants in consensus protocols to filter malicious traffic and improve liveness in presence of DoS attacks.

By definition, *DoS attacks* aim at denying access to *shared* computational and network resources by correct processes [13]. They can address different levels of the protocol stack [23]. If an attacker is able to deny access to the services on a certain level, no upper level protocols will be able to provide the desired services. For example, if an attacker is able to flood the network at the physical level, any OS- or application-level policy aimed at guaranteeing fair access to the resources of the host will be vain. Therefore, the absence of DoS attacks at the lower level of the stack is necessary for successful handling of higher level attacks. Furthermore, higher level functionalities are in general more complex and require more host resources than lower level ones, e.g., a complex database query requires more resources than a opening TCP connection. The amount of traffic an attacker needs to generate to launch DoS attacks using higher level functionalities is generally lower. These attacks can often be prevented by design, for example by authenticating the clients and fairly partitioning the local resources.

In a fail-heterogeneous architecture, crash-only coordinators can participate to the consensus protocol as filtering elements to accurately recognize and handle malicious traffic in a end-to-end manner on all the protocol stack up to the state machine replication level. This overcomes the major limitation of network level detection mechanism, i.e., the lack of information on the expected communication patterns. In HeterTrust, coordinators ensure that clients send at most one request at a time, while servers are prevented from any interaction with other servers for retrieving chosen request, as in previous approaches [21, 29]. The use of crash-only, trusted filtering nodes (such as routers and firewalls) to mitigate DoS attacks is consistent with the network level defences proposed in literature [24], as well as with current commercial solutions. In fact, if such filtering nodes could fail-Byzantine, they could collude with malicious parties to allow (and possibly generate) malicious traffic. However, this does not constitute a complete defence against DoS attacks, as service specific DoS attacks can still be launched, and as any filtering could be invalidated by brute force attacks which deplete network resources by flooding.

In closed and controlled networks, such as LANs, brute force attacks could be prevented by eliminating network-level shared resources (e.g., network links and interfaces) and by instead establishing dedicated links between protocol participants, possibly including clients and coordinators. Furthermore, replication and design diversity can guarantee that only a subset of the server replicas have vulnerabilities to service-level attacks. In this case, if at most d correct servers can be indefinitely delayed by semantic service level attacks, HeterTrust can still achieve liveness under DoS attacks as long as coordinators can receive EXECUTED messages from at least $f + 1$ correct servers, i.e., $s - d > 2f + 1$. In general, other fail-Byzantine state machine protocols which do not rely on fail-crash coordinators for safety and efficiency can take advantage of them for liveness under DoS attacks. According to the lower bound of [18], liveness would be possible in a fail-Byzantine model in presence of up to f malicious process and d correct but indefinitely delayed process (which are equivalent to crashed) if $n > 3f + 2d$.

5.2. Group Membership

Group membership protocols provide a consistent view of the set of processes of the systems that are operational and fully connected (the so-called stable component [7]). They allow designers or distributed protocols to abstract failure detection mechanisms, relying on notifications of view updates to guarantee liveness in presence of node failures or network partitioning. If a process detects that a link to another process is down, either due to network failures or due to process crashes, it initiates a view change to eliminate that process and maintain full connectivity.

As many other fail-crash tolerant protocols, group membership has been adapted to operate in Byzantine environments (as in [10]). However, in order to prevent malicious nodes from triggering isolations of correct nodes it is necessary that a process is accused by at least $f + 1$ other processes, where f is the number of Byzantine nodes, before a view change is initiated. Therefore, up to f correct nodes could be prevented, by network faults, from communicating with up to f other correct parties still without ever updating the view of the stable component. Such faults are generally masked as malicious faults by expensive fail-Byzantine

broadcast primitives, although they represent a much more trivial and common connectivity problem.

A membership service using a fail-heterogeneous architecture could rely on a subset of dedicated coordinators to keep the updated consistent membership view. If a node A cannot receive messages from B and thus suspects it, it can ask the coordinators to act as relay and to route messages among A and B, using protocol-level knowledge to detect faulty nodes in a trustworthy manner. This ensures that either communication between A and B takes place as specified, or consistent isolation of nodes without full connectivity is carried out by the coordination nodes.

6. Conclusions

In this paper we have introduced a new fail-heterogeneous architectural model, which represents an intermediate step between benign fail-crash models and conservative fail-Byzantine models. It is based on a separation of concerns between unconstrained execution nodes and lightweight coordination nodes, with reduced functionalities and thus restricted failure mode. Taking the trustworthy state machine replication problem as an example, we have shown how new solutions under the new model can be developed that keep many advantages of fail-crash protocols, while tolerating more severe failures at the server nodes providing the service of interest. For example, our HeterTrust protocol allows an efficient communication pattern similar to a crash-only protocol, but still ensures properties, such as confidentiality, that are extremely expensive to provide in a homogeneous fail-Byzantine model. Last but not least, HeterTrust reduces the number of required replicas with diversified design. Overall, we believe that the model opens up new possibilities for practical protocols and architectures, addressing multiple problems, that are both efficient and resistant to severe failure modes.

References

[1] M.K. Aguilera *et. al.*, "On Implementing Omega with Weak Reliability and Synchrony Assumptions (Extended Abstract)," *ACM PODC'03*, pp. 306–314, 2003.

[2] Y. Amir *et. al.*, "Customizable Fault Tolerance for Wide-Area Byzantine Replication," *Tech. Rep. CNDS-2006-3*, Johns Hopkins Univ., 2006.

[3] G. Bracha and S. Toueg, " Asynchronous Consensus and Broadcast Protocols," *J. of the ACM*, 32(4), pp. 824-840, Oct. 1985

[4] F.V. Brasileiro *et. al.*, "Implementing Fail-Silent Nodes for Distributed Systems," *IEEE TOC*, 45(11), pp. 1226–1238, Nov. 1996.

[5] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM TOCS*, 20(4), pp. 398–461, Nov. 2002.

[6] M. Castro *et. al.*, "BASE: Using Abstraction to Improve Fault Tolerance," *ACM TOCS*, 21(3), pp. 236–269, Aug. 2003.

[7] G. Chockler *et. al.*, "Group Communication Specifications: A Comprehensive Study," *ACM Comp. Surv.*, 33(4), pp. 427–469, Dec. 2001.

[8] M. Correia *et. al.*, "The Design of a COTS Real-Time Distributed Security Kernel," *EDCC 4*, pp. 234–252, 2004.

[9] M. Correia *et. al.*, " How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," *IEEE SRDS*, pp. 174–183, 2004.

[10] V. Drabkin *et. al.*, " Practical Byzantine Group Communication," *IEEE ICDCS*, pp. 36–46, 2006.

[11] C. Dwork *et. al.*, "Consensus in the Presence of Partial Synchrony," *J. of the ACM*, 35(2), Apr. 1988.

[12] M. Fisher *et.al.*, "Impossibility of Distributed Consensus with One Faulty Process," *J. of the ACM*, 32(2), pp. 374–382, Apr. 1985.

[13] V.D. Gilgor, "A Note on Denial-of-Service in Operating Systems," *IEEE TSE*, 10(3), pp. 320–324, May 1984.

[14] B. Kemme *et.al.*, "Using Optimistic Atomic Broadcast in Transaction Processing Systems," *IEEE TKDE*, 15(4), pp. 1018–1032, Jul. 2003.

[15] L. Lamport, "The Byzantine Generals Problem," *ACM TPLS*, 4(3), pp. 382–401, Jul. 1982.

[16] L. Lamport, "The Part-Time Parliament," *ACM TOCS*, 16(2), pp. 133-169, May 1998.

[17] L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, 32(4), pp. 18–25, Dec. 2001.

[18] L. Lamport, "Lower Bounds for Asynchronous Consensus," *FuDiCo workshop*, pp. 22-23, 2003.

[19] B. Littlewood *et. al.*, "Modelling Software Design Diversity: A Review," *ACM Comp. Surv.*, 33(2), pp. 177–208, Jun. 2001.

[20] C. Marchetti *et. al.*, "Fully Distributed Three-Tier Active Software Replication," *IEEE TPDS*, 17(7), pp. 633-645, Jul. 2006.

[21] J-P. Martin and L. Alvisi, "Fast Byzantine Consensus," *IEEE TDSC*, 3(3), pp. 202–215, Jul. 2006.

[22] F.J. Meyer and D.K. Pradahan, "Consensus with Dual Failure Models," *IEEE TPDS*, 2(2), pp. 214–222, Apr. 1991.

[23] J. Mirkovic and P. Reiher, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms," *ACM SIGCOMM Comp. Comm. Review*, 34(2), pp. 39–53, Apr. 2004.

[24] T. Peng *et.al.*, " Survey of Network-based Defense Mechanisms Countering the DoS and DDoS Problems," *ACM Comp. Surv.*, 30(1), Apr. 2007.

[25] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM TOCS*, 2(2), pp. 145–154, May 1984.

[26] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comp. Surv.*, 22(4), pp. 299–319, Dec. 1990.

[27] P. Verissimo, "Travelling through Wormholes: a New Look at Distributed Systems Models," *ACM SIGACT News*, 37(1), pp. 66–81, 2006.

[28] C. Walter *et. al.*, "Continual On-line Diagnosis of Hybrid Faults," *DCCA-4*, pp. 233–249, 1995.

[29] J. Yin et.al., "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *ACM SOSP*, pp. 253–267, 2003.