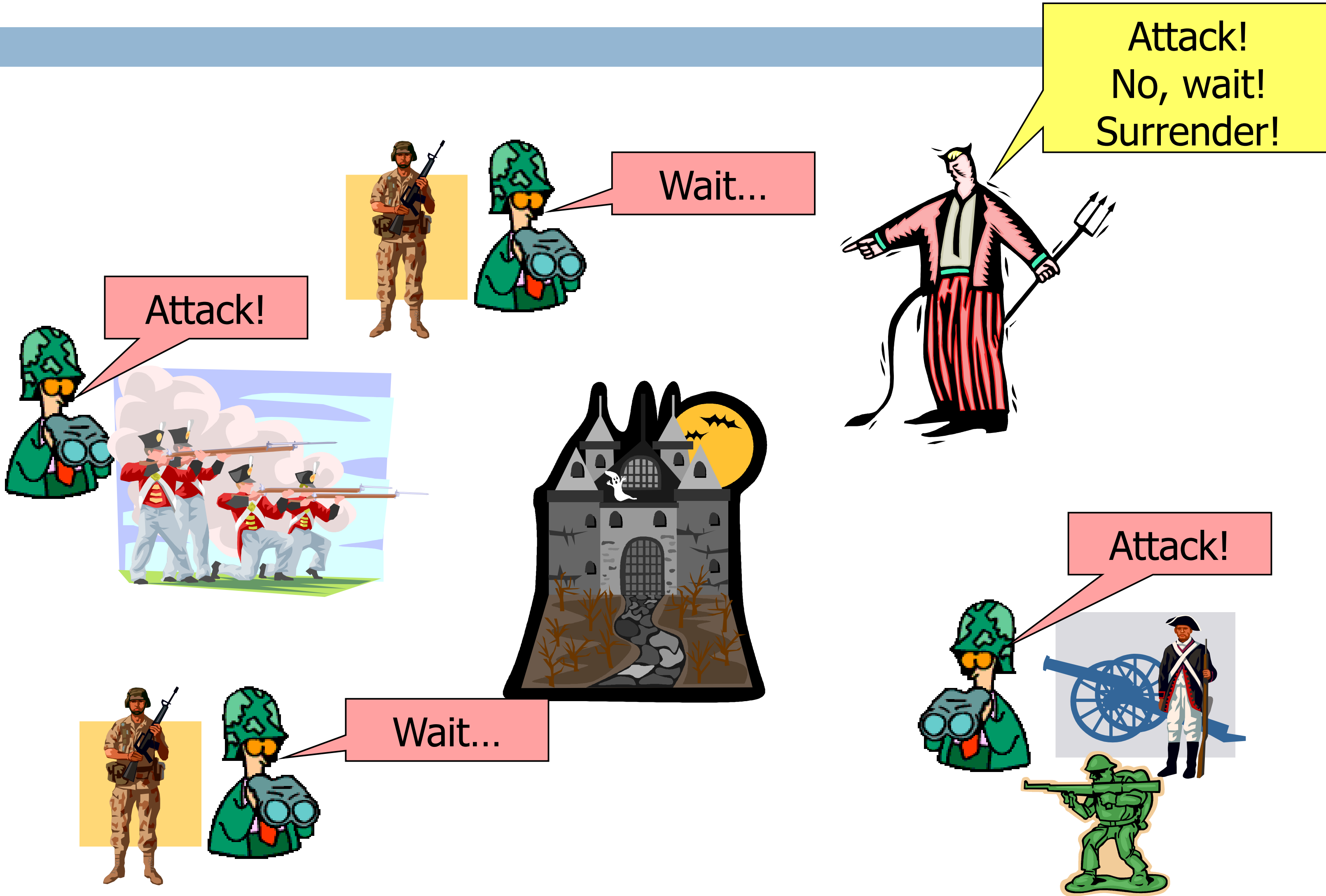
PRACTICAL BYZANTINE FAULT TOLERANCE

(THE BYZANTINE GENERALS PROBLEM)

The Byzantine Generals Problem (Lamport, Shostak, Pease, 1982)

- *The setting:* There are n generals, one of them is the commanding general. Generals can send (and receive messages from other generals)
- *The problem:* Develop a protocol for the commanding general to send an order to his n-1 lieutenant generals such that
  -IC1. All loyal lieutenants obey the same order.
  -IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.
- *The adversary:* Any of the generals could be traitors, i.e., could send inconsistent messages regarding the order to the other generals
- Note nuanced difference from consensus problem

The Byzantine Generals Problem



Impossibility with 3 generals, 1 traitor

- For n=3, m=1 there is no solution

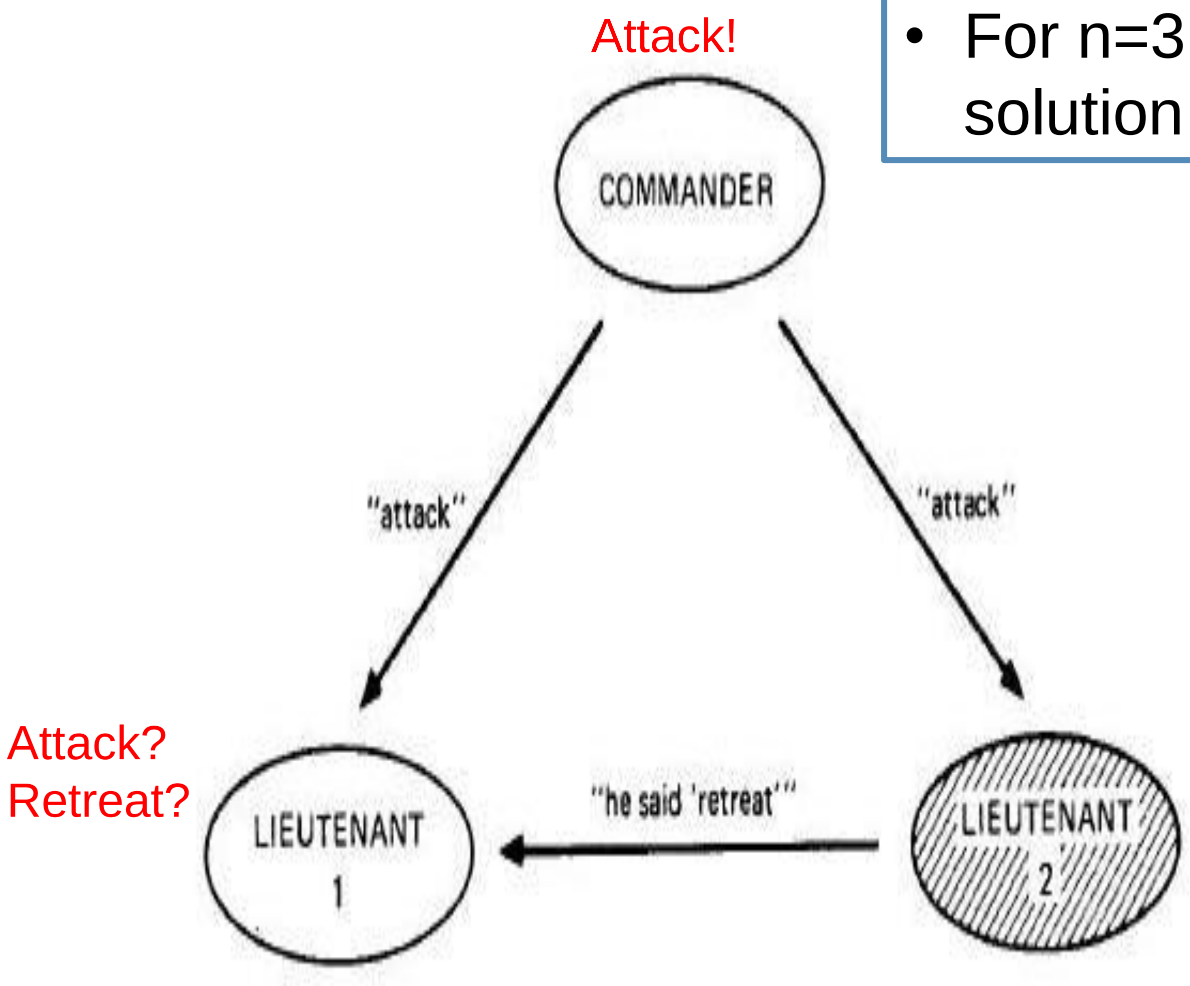


Fig. 1. Lieutenant 2 a traitor.

Impossibility Results

- For $n = 3$ generals and 1 traitor, there is no solution (protocol). This is because a loyal lieutenant cannot distinguish who is the traitor when he gets conflicting information from the commander and the other lieutenant. Let's call this the *3-Generals Problem*.
- BGP for $n < 3m+1$ generals and m traitors can be reduced to the 3 - generals problem, with each of the Byzantine generals simulating at most m lieutenants and taking the same decision as the loyal lieutenants they simulate. Thus BGP for $n < 3m+1$ and m traitors is not solvable.
- Reaching approximation is as hard as reaching agreement.

A Solution with oral messages for n>3m

- A solution for BGP with n>3m nodes and up to m traitors, is given
- *Oral message system properties:*
  - A1. Every message that is sent is delivered correctly. -> No message loss.
  - A2. The receiver of a message knows who sent it. -> Completely connected network with reliable links(due to A1).
  - A3. The absence of a message can be detected. -> Synchronous system only.
- Every general can send a message to every other general.

A Solution with oral messages for n>3m

- *Solution in brief:*
  - uses a function "majority" which takes in a set of values and returns the value that is the majority among them (a possible implementation - median of the values).
  - uses 'rounds' in each of which a general broadcasts the value he has received in the earlier round to all the other generals through whom the value has not passed before he received it.
  - when returning from the round, for each j, any two loyal lieutenants receive the same vector of values {v1, ... v(n-1)}. As the majority of the loyal lieutenants' values in these is ensured, applying the majority function on {v1, ... v(n-1)} to obtain vn preserves the above fact (that any two loyal lieutenants receive the same vector of values {v1, ... vn}). This ensures that BGP is solved.
- *Note:* If the commander is not a traitor, we can be done in 2 rounds. If the commander is a traitor, you may need up to m+1 rounds.

BGP Solution with Oral Messages

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM$(m - 1)$ to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM$(m - 1)$), or else RETREAT if he received no such value. Lieutenant i uses the value $majority(v_1, \ldots, v_{n-1})$.

A solution with (unforgable) signed messages

- The difficulty of BGP is in the ability of a traitor lieutenant to lie about the commander's order.
  - If we can restrict this ability, BGP is solvable *with any number of traitors as long as their maximum number is known.*
- *Signed messages:*
  - Extra A4 assumption needed in addition to the 3 assumptions made in the solution with oral messages
    - A loyal general's signature cannot be forged, any alteration can be detected. This means a traitor can drop a message, but can't change it
    - Any one can verify the authenticity of a signature. This means that no one can fool a general
- Again, assume a fully connected message graph among the generals.

A solution with (unforgable) signed messages with *m* traitors and any *n* generals

- ***Solution in brief:***
  - Uses a majority-like function called choice.
  - The commander sends a signed order to lieutenants
  - If a lieutenant receives an order from someone (either from commander directly, or from other lieutenants), he verifies it and then puts it in a set V if it's not already there. Relay the order if there are less than m *distinct* signatures on the order.
  - Everyone halts at round m+1, and uses choice(V) as the desired action

A solution with (unforgable) signed messages with *m* traitors and any *n* generals

- The algorithm is to make all loyal lieutenants keep the same set of V, thus choice(V) is the same.
- If the commander is loyal, all loyal lieutenants have the correct order by round 1 and by unforgablity no more orders can be produced.
- If the commander is not loyal, by running the algorithm to round m+1, at least one loyal lieutenant will get the order before round m (because there are only m traitors). And that loyal lieutenant will send it to all others. In short, if one loyal lieutenant gets an order, all loyal lieutenants will get it in the next round.

A solution with (unforgable) signed messages with *m* traitors and any *n* generals

Algorithm SM(m).

Initially $V_i = \emptyset$.

(1) The commander signs and sends his value to every lieutenant.

(2) For each i:

(A) If Lieutenant i receives a message of the form $v:0$ from the commander and he has not yet received any order, then

(i) he lets V_i equal $\{v\}$;

(ii) he sends the message $v:0:i$ to every other lieutenant.

(B) If Lieutenant i receives a message of the form $v:0:j_1:\cdots:j_k$ and v is not in the set V_i, then

(i) he adds v to V_i;

(ii) if $k < m$, then he sends the message $v:0:j_1:\cdots:j_k:i$ to every lieutenant other than $j_1, \ldots, j_k$.

(3) For each i: When Lieutenant i will receive no more messages, he obeys the order $choice(V_i)$.

BGP Theorems

- *Theorem 1. For any m, Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more than 3m generals and at most m traitors*
- Theorem 2. For any m, Algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors
- Both require message paths of length up to m+1 (very expensive)
- Both require that absence of messages must be detected (A3) via time-out (vulnerable to DoS)

Relaxing the assumption on full-connectivity

- Previous 2 solutions can be extended to relax the assumption that the message graph among the generals is fully connected.
- *Oral messages:* Solution with oral messages is extended to solve BGP with up to m traitors in a p-regular graph with $m>0$ and $p>3m-1$.
- *Unforgable messages:* Can solve BGP with up to m traitors in $(m+d-1)$ rounds, where d is the diameter of the subgraph of loyal generals.
  - Assumption: subgraph of loyal generals is connected (this can be relaxed by relaxing the problem statement of BGP)

Practical use of BGP in real world systems

- The best way to provide fault-tolerant decision-making in redundant systems is by majority voting.
  - A faulty input device may generate meaningless inputs, but majority voting would ensure that the **same** meaningless values are used.
- For majority voting to yield a reliable system, the following 2 conditions must be satisfied
  - All non-faulty processors must use the same input value
  - If input unit is non-faulty, then all non-faulty processes use the value it provides
- But these are just the requirements of the BGP!
  - So we can apply the above solutions to the BGP in real-life

Practicality of assumptions made?

- A1. Every message that is sent is delivered correctly. This means no message loss.
  - In real life, link failures occur. However, link failures are indistinguishable from failures of processors, therefore we can count the link failures as one of the *m*.
  - Signed message is insensitive to link failures because no message can be forged even if links fail.
- A2. The receiver of a message knows who sent it. This means we have a completely connected network with reliable links (due to A1).
  - What is actually required is that no traitor can forge a non-faulty process' message.

Practicality of assumptions made?

- A3. The absence of a message can be detected. This means we have a synchronous system only.
  - In an asynchronous system, this condition cannot be satisfied. It is usually implemented via time-outs.
- A4. A loyal general's signature cannot be forged, any alteration can be detected.
  - If processor is non-faulty, then no faulty processor can generate S(M). This can never be completely guaranteed, but its probability can be reduced
  - Given M and X, any one can verify if X == S(M). This is doable in real world.

Questions

- *Graph connectivity.* Are p-regular topologies that frequent ? Can we extend the BGP solutions to any network topology ? Has it been extended to any other topologies ?
- *Value of m:* How would one obtain a reasonable value for maximum m in a practical system (note that this maximum number is required even in the solution with signed messages).
- *Synchronous/asynchronous systems:* How many synchronous system do we really use (SMP machines, and?) How about asynchronous systems ?

Questions

- *Further work after this paper:*
  - What other solutions to BGP have been proposed after this paper ?
  - Has any attempt been made to extend the BGP solutions to asynchronous systems to ensure 'some degree/probability' of reliability ?
  - References on next slide
- Bounds on best possible BGP solution (in terms of messages) ?

Related follow-on work

- *Impossibility/necessity results*
  - Fischer, M. J., Lynch, N. A., and Paterson, M. S. ``Impossibility of Distributed Consensus with One Faulty Process," *J. ACM 32*, 2 (April 1985), 374--382.
  - Dolev, D., Dwork, C., and Stockmeyer, L. ``On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM 34*, 1 (January 1987), *77--97.*
- *Approximate agreement*
  - Bracha, G. ``An O(log n) Expected Rounds Randomized Byzantine Generals Protocol," *J. ACM 34*, 4 (October 1987), 910--920.
  - Bracha, G. and Toueg, S. ``Asynchronous Consensus and Broadcast Protocols," *J. ACM 32*, 4 (October 1985), 824--840.
  - Ben-Or, M. ``Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *ACM Symposium on Principles of Distributed Computing*, 1983, 27--30.

Related follow-on work

- *Approximate agreement (cont'd)*
  - Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. ``Reaching Approximate Agreement in the Presence of Faults," *J. ACM* 33, 3 (July 1986), 499--516.
  - Dolev, D., Ruediger, R., and Strong, H. R. ``Early Stopping in Byzantine Agreement," *J. ACM 37*, 4 (October 1990), 720--741.
  - Hadzilacos, V. and Halpern, J. Y. ``Message-Optimal Protocols for Byzantine Agreement," *ACM Symposium on Principles of Distributed Computing*, 1991, 309--323.
  - Halpern, J. Y., Moses, Y., and Waarts, O. ``A Characterization of Eventual Byzantine Agreement," *ACM Symposium on Principles of Distributed Computing*, 1990, 333--346.

Related follow-on work

- *Failure detectors*
  - Chandra, T. D., Hadzilacos, V., and Toueg, S. ``The Weakest Failure Detector for Solving Consensus," *ACM Symposium on Principles of Distributed Computing*, 1992, 147--158.
  - Chandra, T. D. and Toueg, S. ``Unreliable Failure Detectors for Asynchronous Systems," *ACM Symposium on Principles of Distributed Computing*, 1991, 325--340.

Break

Practical Byzantine Fault Tolerance

- Malicious attacks and software errors that can cause arbitrary behaviors of faulty nodes are increasingly common
- Previous solutions assumed synchronous system and/or were too slow to be practical
  - e.g. Rampart, OM, SM
- This paper describes a new replication algorithm that tolerates Byzantine faults and is practical
  - asynchronous environment, better performance

PBFT System Model

- Asynchronous distributed system where nodes are connected by a network
- Byzantine failure model
  - faulty nodes behave arbitrarily
  - independent node failures
- Cryptographic techniques to prevent spoofing and replays and to detect corrupted messages
- Very strong adversary

Service Properties

- Any deterministic replicated service with a state and some operations
- Assuming less than one-third of replicas are faulty
  - safety (linearizability)
  - liveness (assuming delay(t) >> t)
- Access control to guard against faulty client
- The resiliency (3f+1) of this algorithm is proven to be optimal for an asynchronous system

The Algorithm

- Basic setup:

  - $|\mathcal{R}| = 3f + 1$

  - A view is a configuration of replicas (a primary and backups): $p = v \bmod |\mathcal{R}|$

  - Each replica is deterministic and starts with the same initial state

  - The state of each replica includes the state of the service, a message log of accepted messages, and a view number

The Algorithm

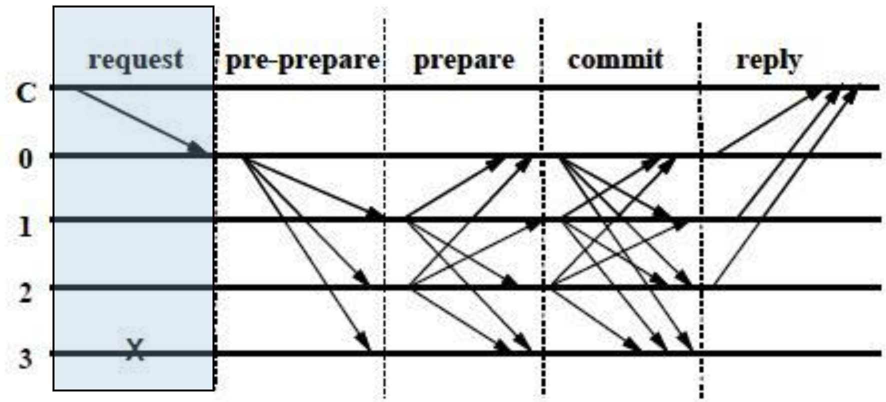


Figure 1: Normal Case Operation

- 1. A client sends a request to invoke a service operation to the primary

$$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$$

o= requested operation
t= timestamp
c= client
ϭ= signature

The Algorithm

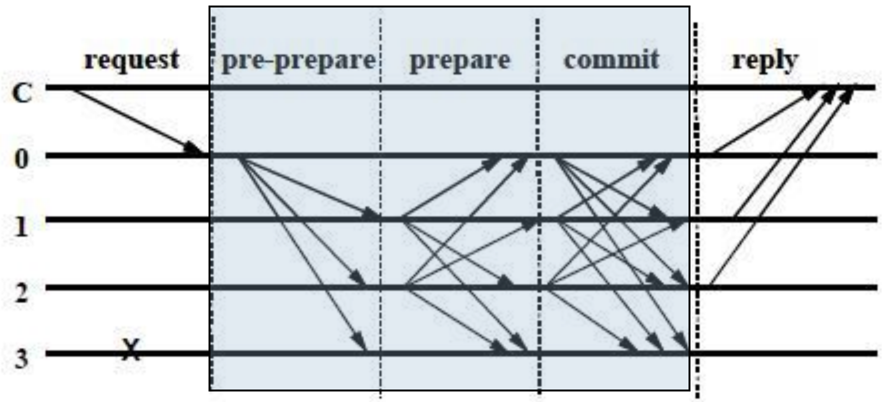


Figure 1: Normal Case Operation

- 2. The primary multicasts the request to the backups (three-phase protocol)

The Algorithm

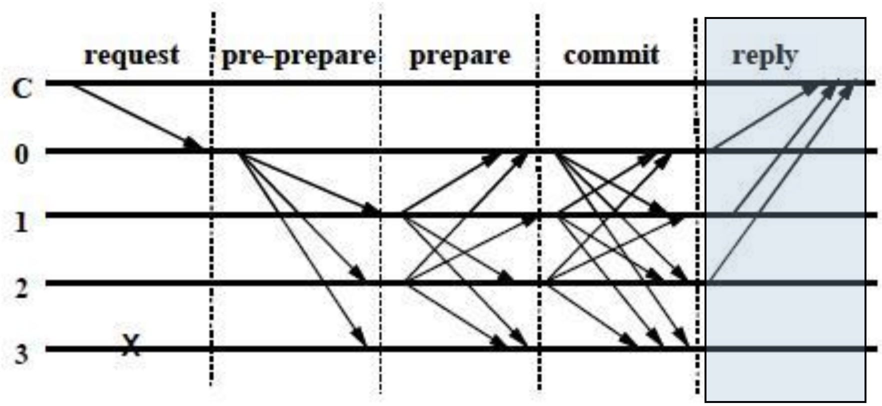


Figure 1: Normal Case Operation

- 3. Replicas execute the request and send a reply to the client

$$\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$$

o= requested operation
t= timestamp
c= client
ϭ= signature

v= view
i= replica
r= result

The Algorithm

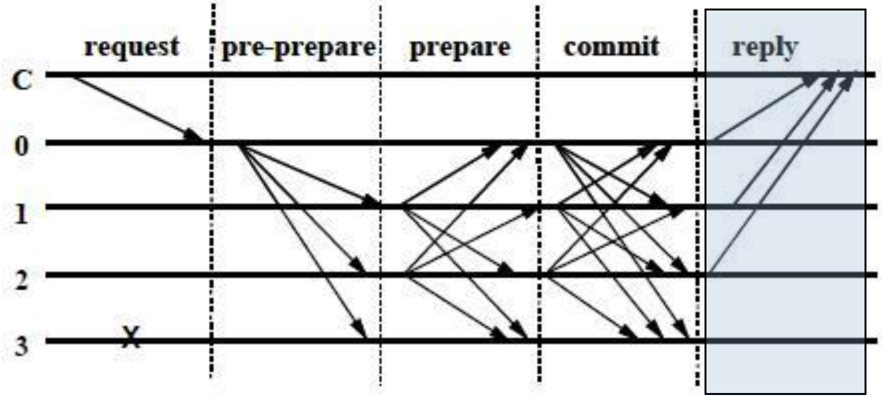


Figure 1: Normal Case Operation

- 4. The client waits for f+1 replies from different replicas with the same result; this is the result of the operation

Three-phase Protocol

- 1.pre-prepare (pp)
  - primary assigns n to the request; multicasts pp
  - request message m is piggy-backed (request itself is not included in pp)
  - accepted by backup if: $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$
    - the messages are properly signed;
    - it is in the same view v;
    - the backup has not accepted a pp for the same v and n with different d
    - h <= n <= H
  - if accepted, then replica i enters prepare phase

Three-phase Protocol

- 2.prepare (p)

  - if backup accepts pp, multicasts p

  - accepted by backup if: $\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$

    - message signature is correct;
    - in the same view;
    - h<= n<= H

  - prepared(m,v,n,i) is true if i has logged:

    - request message m
    - pp for m in v
    - 2f matching prepares with the same (v,n,d)

  - if prepared becomes true, multicasts commit message and enters commit phase

Three-phase Protocol

- Pre-prepare – prepare phases ensure the following invariant:

  - if prepared(m,v,n,i) is true then prepared(m',v,n,j) is false for any non-faulty replica j (inc. i=j) and any m' such that D(m') != D(m)

- i.e. ensures requests in the same view are totally ordered (over all non-faulty replicas)

Three-phase Protocol

- 3.commit
  - accepted by backup if: $\langle \text{COMMIT}, v, n, D(m), i \rangle_{\sigma_i}$
    - message signature is correct;
    - in the same view;
    - h<= n<= H
  - committed(m,v,n) is true iff prepared(m,v,n,i) is true for all i in some set of f+1 non-faulty replicas
  - committed-local(m,v,n,i) is true iff prepared(m,v,n,i) is true and i has accepted 2f+1 matching commits
  - replica i executes the operation requested by m after committed-local(m,v,n,i) is true and i's state reflects the sequential execution of all requests with lower n

Three-phase Protocol

- Commit phase ensures the following invariant:
  - *if committed-local(m,v,n,i) is true for some non-faulty i, then committed(m,v,n) is true*
- i.e. any locally committed request will eventually commit at f+1 or more non-faulty replicas
- The invariant and view change protocol ensure that non-faulty replicas agree on the sequence numbers of requests that commit locally even if they commit in different views at each replica
- Prepare – commit phases ensure requests that commit are totally ordered across views

The Algorithm

- Garbage Collection
  - must ensure safety still holds after discarding messages from log
  - generates checkpoint (a snapshot of the state) periodically
    - checkpoint: multicast checkpoint message with seq number and digest of state
    - if a replica receives 2f+1 matching checkpoint messages, the checkpoint becomes stable and any messages associated with seq numbers less than that of the checkpoint are discarded
- View Changes
  - provides liveness
  - triggered by timeout to prevent backups from waiting forever
  - with commit phase invariant, view change guarantees total ordering of requests across views (by exchanging checkpoint information across views)

The Algorithm

- The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally
- To provide liveness, replicas must change view if they are unable to execute a request
  - avoid view change that is too soon or too late
  - faulty replicas can't force frequent view changes; liveness guaranteed unless message delays grow faster than the timeout period indefinitely

Optimizations

- Reducing Communication
  - avoids sending most of large replies
    - only designated replica sends result
  - reduces number of message delays for an operation invocation from 5 to 4
    - execute a request tentatively if prepared
    - client waits for matching 2f+1 tentative replies
  - improves performance of read-only operation
    - client multicasts a read-only request to all
    - replicas execute it immediately in tentative state
    - send back replies after requests reflected in the tentative state commit
    - client waits for 2f+1 replies with the same result
  - treating small and big requests differently

Optimizations

- Cryptography

  - digital signatures used only for view-change and new-view messages (but view change is not implemented!)

  - authenticate all other messages using message authentication codes (MACs)

Implementation

- The Replication Library

  - basis for any replication service

  - client: invoke

  - server: execute, make_checkpoint, delete_checkpoint, get_digest, get_checkpoint, set_checkpoint

  - point-to-point communication using UDP

  - view change and retransmission can be used to recover from lost messages

  - did not implement view-change or retransmission, but claims this does not compromise the accuracy of the results

Implementation

- A Byzantine-Fault-tolerant File System

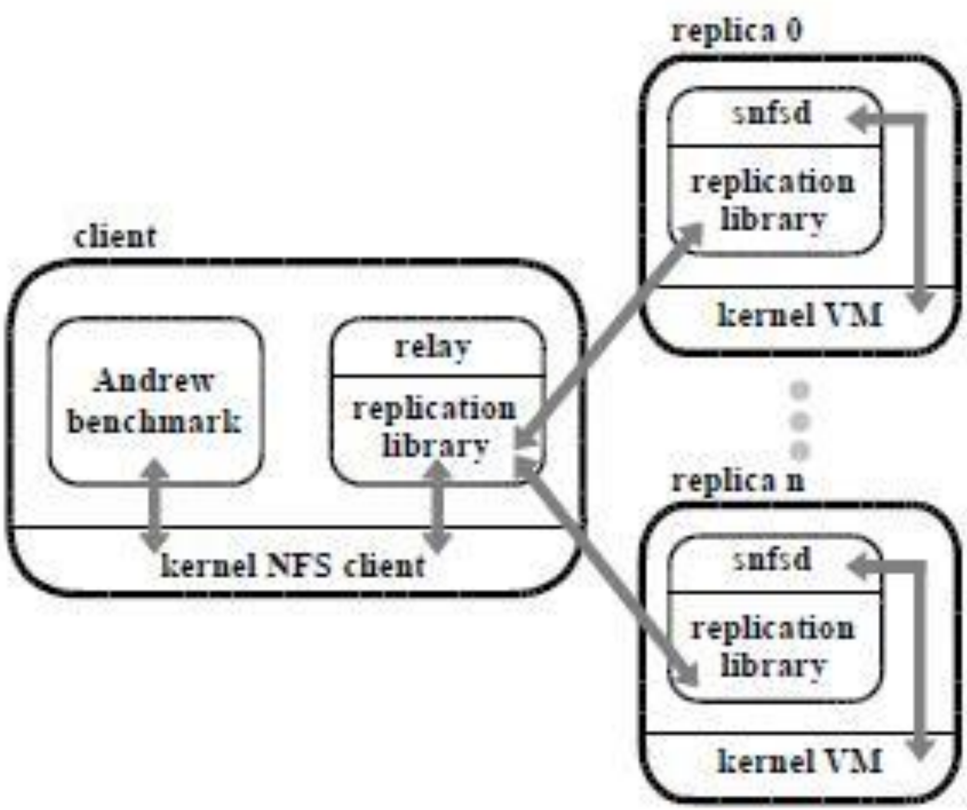


Figure 2: Replicated File System Architecture.

Implementation

- Maintaining Checkpoints
  - snfsd uses direct file system operations on memory mapped file system to preserve locality
  - checkpoint record (n, list of modified blocks, d) that keeps update information for the corresponding checkpoint
  - snfsd keeps a copy-on-write bit for every 512-byte block
  - copy-on-write technique to reduce space and time overhead in maintaining checkpoints

- Computing Checkpoint Digests
  - AdHash: sum of digest of each block (index+value)
  - efficient for a small number of modified blocks

Performance Evaluation

- Micro-benchmark: invoke null-op; provides service independent evaluation of the performance of the replication library
- Andrew-benchmark: emulates a software development workload; compares BFS with NFS V2 and BFS without replication
- Measured normal-case behaviors (i.e. no view changes) in an isolated network with 4 replicas
  - the first correct replicated service in asynchronous environment like internet
  - can tolerate Byzantine faults (liveness) with comparable normal-behavior performance (when there are no faults)

Performance Evaluation

arg./res. (KB)	replicated		without replication
	read-write	read-only	
0/0	3.35 (309%)	1.62 (98%)	0.82
4/0	14.19 (207%)	6.98 (51%)	4.62
0/4	8.01 (72%)	5.94 (27%)	4.66

Table 1: Micro-benchmark results (in milliseconds); the percentage overhead is relative to the unreplicated case.

phase	BFS		BFS-nr
	strict	r/o lookup	
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

Table 2: Andrew benchmark: BFS vs BFS-nr. The times are in seconds.

phase	BFS		NFS-std
	strict	r/o lookup	
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

Table 3: Andrew benchmark: BFS vs NFS-std. The times are in seconds.

Some criticisms

- No mention is made on how the group is actually formed. Is it static or dynamic?
- Pushing checkpointing to the application level makes the application harder. Checkpoints and copy on write seem a must.
  - That's probably why the authors took the memory-mapped file direction for NFS implementation, instead of the much simpler layer over an existing OS file system. This makes it hard to port existing applications to such a platform.
- Storing all application replies to be able to retransmit them to the clients might not be efficient enough.
  - Database appls might have large result-sets and that would put certain space/time requirements on each replica peer.
- The comparison with NFS in not apples-to-apples.

Conclusion

- PBFT is the first replicated system that works correctly in asynchronous system and it improves performance of previous algorithms by more than an order of magnitude
- Prior SMR algorithms were too slow to be used in practice (proportional to the number of faulty nodes vs. number of phases)