Practical Byzantine Fault Tolerance
(The Byzantine Generals Problem)
Introduction

• 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 practical (asynchronous environment, better performance)
• Why PBFT is practical (compared to the solutions from the Byzantine Generals Problem)
The Byzantine Generals Problem

From cs4410 fall 08 lecture
The Byzantine Generals Problem

• A commanding general must 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.
Byzantine Generals Problem

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 $\text{majority}(v_1, \ldots, v_{n-1})$.

Fig. 1. Lieutenant 2 is a traitor.
Byzantine Generals Problem

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 $\text{choice}(V_i)$.
Byzantine Generals Problem

- 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).
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): $p = v \mod |\mathcal{R}|$
  - 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

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

\( \langle \text{REQUEST}, o, t, c \sigma_c \rangle \)

- \( o = \) requested operation
- \( t = \) timestamp
- \( c = \) client
- \( \sigma = \) signature

Figure 1: Normal Case Operation
The Algorithm

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

- 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
- \( v = \) view
- \( t = \) timestamp
- \( i = \) replica
- \( c = \) client
- \( r = \) result
- \( \sigma = \) signature

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**
   - 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: $\langle\text{PRE-PREPARE}, v, n, d, \sigma_p, m \rangle$
     - 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
     - $h \leq n \leq H$
   - If accepted, then replica i enters prepare phase
Three-phase Protocol

- **2.prepare**
  - if backup accepts pp, multicasts p
  - accepted by backup if:
    - message signature is correct;
    - in the same view;
    - $h \leq n \leq H$
  - $\text{prepared}(m,v,n,i)$ 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 \( \text{prepared}(m,v,n,i) \) is true then \( \text{prepared}(m',v,n,j) \) is false for any non-faulty replica \( j \) (inc. \( i=j \)) and any \( m' \) such that \( D(m') \neq D(m) \)

- 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: \(<\text{COMMIT}, v, n, D(m), i>_{\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 the safety still holds after discarding messages from log
  - generates checkpoint (a snapshot of the state) every once in a while
  - when a replica generates a checkpoint, it multicasts 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
  - timer starts when backup receives a valid request; it stops when the replica is no longer waiting to execute the request
  - 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 changes that are too soon or too late; the fact that faulty replicas can’t force frequent view changes will guarantee liveness unless message delays grow faster than the timeout period indefinitely.
Optimizations

- Reducing Communication
  - avoids sending most of large replies
  - only designated replica send the result
  - reduces the 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 the performance of read-only operations
    - 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
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
  - It does not implement view-change or retransmission at present, but this does not compromise the accuracy of the results
Implementation

- A Byzantine-Fault-tolerant File System
Implementation

• Maintaining Checkpoints
  - snfsd uses direct file system operations on memory mapped file system to preserve locality
  - checkpoint record \((n, \text{list of modified blocks}, d)\) 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 \((\text{index}+\text{value})\)
  - 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?!
Performance Evaluation

<table>
<thead>
<tr>
<th>arg./res. (KB)</th>
<th>replicated</th>
<th>without replication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read-write</td>
<td>read-only</td>
</tr>
<tr>
<td>0/0</td>
<td>3.35 (309%)</td>
<td>1.62 (98%)</td>
</tr>
<tr>
<td>4/0</td>
<td>14.19 (207%)</td>
<td>6.98 (51%)</td>
</tr>
<tr>
<td>0/4</td>
<td>8.01 (72%)</td>
<td>5.94 (27%)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>phase</th>
<th>BFS</th>
<th>BFS-nr</th>
<th>BFS</th>
<th>NFS-std</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>strict</td>
<td>r/o lookup</td>
<td>strict</td>
<td>r/o lookup</td>
</tr>
<tr>
<td>1</td>
<td>0.55 (57%)</td>
<td>0.47 (34%)</td>
<td>0.35</td>
<td>0.47 (-73%)</td>
</tr>
<tr>
<td>2</td>
<td>9.24 (82%)</td>
<td>7.91 (59%)</td>
<td>5.08</td>
<td>7.91 (-16%)</td>
</tr>
<tr>
<td>3</td>
<td>7.24 (18%)</td>
<td>6.45 (6%)</td>
<td>6.11</td>
<td>6.45 (20%)</td>
</tr>
<tr>
<td>4</td>
<td>8.77 (18%)</td>
<td>7.87 (6%)</td>
<td>7.41</td>
<td>7.87 (19%)</td>
</tr>
<tr>
<td>5</td>
<td>38.68 (20%)</td>
<td>38.38 (19%)</td>
<td>32.12</td>
<td>38.38 (-2%)</td>
</tr>
<tr>
<td>total</td>
<td>64.48 (26%)</td>
<td>61.07 (20%)</td>
<td>51.07</td>
<td>61.07 (-2%)</td>
</tr>
</tbody>
</table>

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

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

• PBFT is the first replicated system that works correctly in asynchronous system like internet and it improves performance of previous algorithms by more than an order of magnitude

• OM-SM algorithms are too slow to be used in practical (proportional to the number of faulty nodes vs. number of phases)