Practical Byzantine Fault Tolerance

Castro and Liskov, OSDI 1999

Nathan Baker, presenting on 23 September 2005

Practical Byzantine Fault Tolerance

- What is a Byzantine fault?
- Rationale for Byzantine Fault Tolerance
- BFT Algorithm
- Conclusion

What is a Byzantine fault?

- Arbitrary node behavior
  - Failure to return a result
  - Return of an incorrect result
  - Return of a deliberately misleading result
  - Return of a differing result to different parts of the system
- Source: Byzantine Generals Problem, Lamport, Shostak, and Pease (1982)

Rationale for BFT

- Guard against malicious attacks
- Prevent faulty code at a single node from corrupting the system
- Ultimate goal: provide system consistency even when nodes may be inconsistent
- Useful in distributed areas like file servers or automated control systems where state is very important

Overview of Solution

- *n* generals need to achieve consensus
- *f generals may be traitors*
- *Consider a voting algorithm*
  - *If a general sees f + 1 identical responses, that response must be correct*

Simple Example

- Consider four replicas trying to agree on the value of a single bit (attack/don't attack)

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1			
Replica 2		1		
Replica 3			1	
Replica 4				0

Simple Example

- All replicas send their value to the other replicas

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	1	1	0
Replica 2	1	1	1	0
Replica 3	1	1	1	0
Replica 4	1	1	1	0

Simple Example

- Now, all replicas send their entire vector to all other replicas
  - 2 sends values for <2,3,4> to 1 and <1,2,4> to 3

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	<1,1,0>	<1,1,0>	<0,0,0>
Replica 2	<1,1,0>	1	<1,1,0>	<0,0,0>
Replica 3	<1,1,0>	<1,1,0>	1	<0,0,0>
Replica 4	<1,1,1>	<1,1,1>	<1,1,1>	0

Simple Example

- Result is the most frequent value in the vector

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	1	1	0
Replica 2	1	1	1	0
Replica 3	1	1	1	0
Replica 4	1	1	1	0

Simple Example

- Question: in this example we had 4 replicas, one of which was faulty. Would this work with 3 replicas, one of which was faulty?
  - Hint: this is an asynchronous environment

BFT Algorithm

- Algorithm discussion
  - Overview
  - Details
  - BFS
  - Evaluation

BFT Algorithm Overview

- Previous work was slow-running or relied on synchrony for safety.
- This algorithm (BFT) provides safety and liveness over an asynchronous network.
  - Safety: the system maintains state and looks to the client like a non-replicated remote service. Safety includes a total ordering of requests.
  - Liveness: clients will eventually receive a reply to every request sent, provided the network is functioning.

BFT Algorithm Overview

- Based on state machine replication
- Messages signed by public key cryptography
- Message digests created using collision-resistant hash functions
- Uses consensus and propagation of system views: state is only modified when the functioning replicas agree on the change

BFT Algorithm Overview

- For *n clients, there are n 'views', {0..n-1}.*
  - *In view i, node i is the primary node*
  - *View change is increment mod n*
  - *View change occurs when 2f nodes believe the primary has failed*
- *Guaranteed safety and liveness provided less than* $\left\lfloor \frac{n-1}{3} \right\rfloor$ *= f replicas have failed.*

BFT Algorithm: Normal Operation

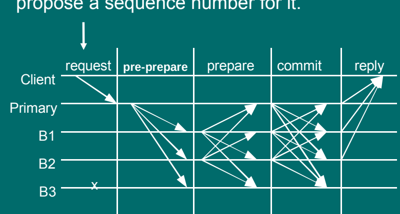
1. The client sends a request to the primary.
2. The primary assigns the request a sequence number and broadcasts this to all replicas (pre-prepare).
3. The replicas acknowledge this sequence number (prepare).
4. Once 2*f prepares have been received, a client broadcasts acceptance of the request (commit).*

BFT Algorithm: Normal Operation

5. Once 2*f +1 commits have been received, a client places the request in the queue.*
   5.1. *In a non-faulty client, the request queue will be totally ordered by sequence number.*
6. *Once all prior requests have been completed, the request will be executed and the result sent directly to the client.*
7. *All these messages are logged.*

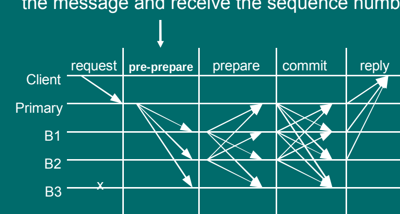
BFT Algorithm: Normal Operation

Phase 1: Client sends a request to the primary. The primary can then validate the message and propose a sequence number for it.



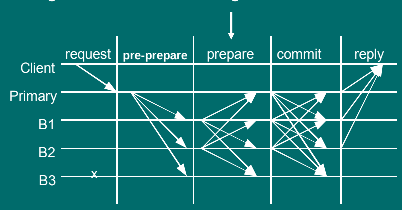
BFT Algorithm: Normal Operation

Phase 2: Primary sends pre-prepare message to all backups. This allows the backups to validate the message and receive the sequence number.



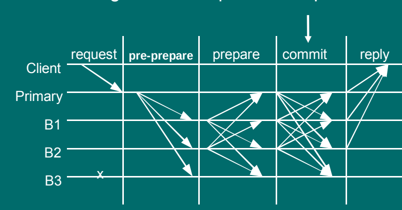
BFT Algorithm: Normal Operation

Phase 3: All functioning backups send prepare message to all other backups. This allows replicas to agree on a total ordering.



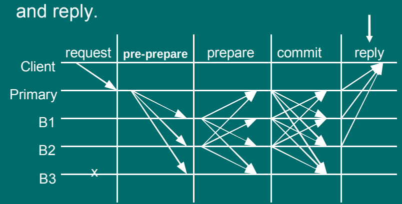
BFT Algorithm: Normal Operation

Phase 4: All replicas multicast a commit. The replicas have agreed on an ordering and have acknowledged the receipt of the request.



BFT Algorithm: Normal Operation

Phase 5: Each functioning replica sends a reply directly to the client. This bypasses the case where the primary fails between request and reply.



BFT Algorithm: View Changes

- What if the primary is faulty?
  - The client uses a timeout. When this timeout expires, the request is sent to all replicas.
  - If a replica already knows about the request, the rebroadcast is ignored.
  - If the replica does not know about the request, it will start a timer.
  - On timeout of this second timer, the replica starts the view change process.

BFT Algorithm: View Changes

- If a replica's timer expires, it sends a view change message.
  - This message contains the system state (in the form of archived messages) so that other nodes will know that the replica has not failed.
- If the current view is *v, node v+1 (mod n) waits for 2f valid view-change messages.*

BFT Algorithm: View Changes

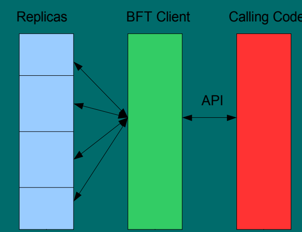
- Once *v+1 has seen 2f view-change messages, it multicasts a new-view message*
  - *This message contains all the valid view change messages received by v+1 as well as a set O of all requests that may not have been completed yet (due to primary failure).*
- *After a replica receives a valid view-change message, it enters view v+1 and processes O*
- *While view change is occurring, no new requests are accepted.*

BFT Algorithm: Client's perspective

- **The client must be BFT-aware:**
  - **Must implement timeout for view-change**
  - **Must wait for replies directly from replicas**
- **The client waits for** *f+1 replies, then accepts the result.*
- *Seamless integration requires three-tier approach.*

25

BFT Algorithm: Client's Perspective

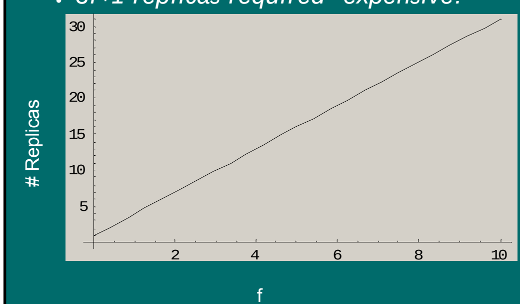


In order to provide seamless interaction with the calling code, there should be a client layer between the replicas and the program.

26

BFT Algorithm: Evaluation

- *3f+1 replicas required--expensive!*



27

BFT Algorithm: Evaluation

- Problems
  - Not scalable
  - Significant overhead
- However
  - Provides Byzantine fault tolerance that can be used in real-world applications

28

uestions?

29

BFT Algorithm: BFS

- The authors implemented a Byzantine Fault-Tolerant NFS system called BFS.
- BFS is comparable in performance to NFS on average, while providing tolerance for Byzantine faults.
- View changes not implemented in BFS.

30

BFT Optimizations

- Optimization
  - Checkpoints/Garbage Collection
  - Reducing communication
  - Message Authentication Codes

BFT Optimizations: Checkpoints

- Pre-prepare, prepare, and commit messages are stored to provide proof of correctness. Storing these messages can be expensive.
- Instead, a checkpoint system is used
  - A checkpoint size *c is set*
  - *A proof of correctness is generated when s mod c = 0*
  - *This is called a checkpoint.*

BFT Optimizations: Checkpoints

- After a checkpoint is produced, a checkpoint message is multicast
- Once 2*f+1 checkpoint messages have been collected, that checkpoint is considered stable and all archived messages with s less than the checkpoint number are discarded. All earlier checkpoints are also discarded.*

Optimizations: Reducing Messages

- This protocol is very message intensive, but there are three ways it can be altered to limit traffic:
- Single result
  - The client request designates only one replica to send the result, and others just send digests
  - If the correct result is not received, the client requests that all nodes send the replies.
  - Only useful for large replies

Optimizations: Reducing Messages

- Tentative replies
  - If a replica's queue is empty, it can compute the result upon the receipt of 2*f prepare messages.*
  - *The replicas then send these tentative replies to the client.*
  - *If the client receives 2f+1 matching tentative replies, this is equivalent to a commit. If not, it retransmits the request and waits for f+1 committed replies.*

Optimizations: Reducing Messages

- Read-only requests
  - The client can transmit read-only requests directly to all replicas.
  - After verifying the request, the reply can be processed and sent directly to the client.
  - If the client receives 2*f+1 identical replies, it accepts the result.*
  - *If not, it retransmits the request as a normal (read/write) operation.*