Tutorial on
EuroSys 2012

(BFT) State Machine Replication:
The Hype, The Virtue... and even some Practice

Alysson Neves Bessani
bessani@di.fc.ul.pt
University of Lisbon, Faculty of Sciences

 1

Summary

- Part 1: The Basics
  - State machine replication
  - Potential applications
  - 5 fundamental results on distributed systems
  - Paxos/Viewstamped replication
  - Castro & Liskov' PBFT
- Part 2: BFT Literature Review
  - Improving performance
  - Improving resource efficiency
  - Improving robustness
- Part 3: Applications, Open Problems & Practice
  - BFT Applications
  - Open problems on BFT
  - BFT-SMaRt
  - Practice: a BFT KV (in memory) Store

 2

Part I

The Basics

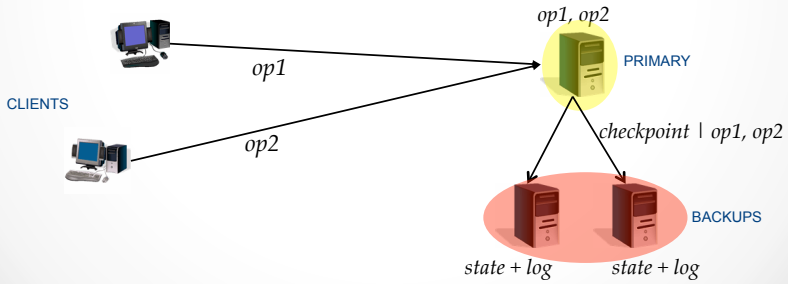
EuroSys 2012

Replication

- Replication is a technique used for **performance** and/or **fault tolerance**
- Each replica is a state machine:
  - A deterministic program that receives an input, change its state and produces an output
  - State transitions are atomic
- Replication can be **passive** or **active**

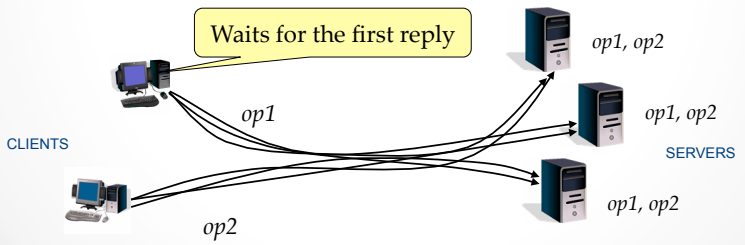
Passive Replication

- Also called **Primary-Backup** (PB) or master-slave
- Clients talk with the primary, that sends the operations and checkpoints to the backups
  - Sometimes backup replicas answer read-only operations
- If the primary crashes, one of the backups takeover



Active Replication

- Also called **State Machine Replication** – SMR (Schneider, ACM CS 1990)
- All servers execute the same set of operations in the same order (servers are always "synchronized")
- Clients wait for the first reply (crash faults)



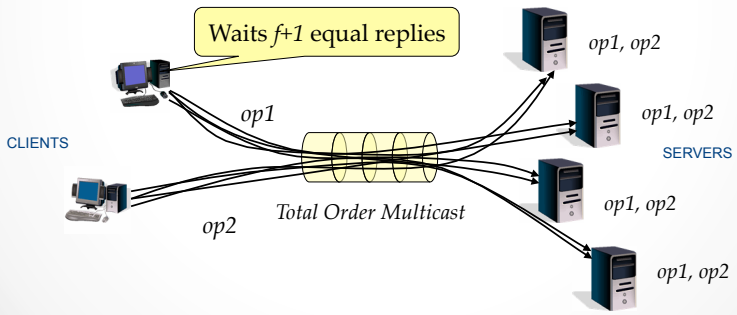
Byzantine Fault Tolerance

- A component that suffers an arbitrary (or Byzantine) failure can exhibit any behavior
  - Stay silent, delay messages, change messages
  - It can model intrusions
- **PB** is hard to use with this fault model
  - **How to know if the reply/delta checkpoint produced by the primary is correct?**
- **SMR** is the way to go:
  - All replicas execute the operations and send replies
  - Clients can vote for the correct reply

 7

BFT State Machine Replication

- All servers execute the same sequence of operations
- Requires total order multicast



 8

SMR Requirements

- **Initial state:** All replicas start on the same state

Easy!

- **Coordination:** All replicas receive the same sequence of inputs

Total Order Multicast

- **Determinism:** all replicas receiving the same input on the state produce the same output and resulting state

Easy?

System Properties

- **Safety**: all servers execute the same sequence of requests
- **Liveness**: all correct clients requests are executed

System Models: BFT SMR Assumptions

- Faults:
  - How many faulty servers and clients the system tolerate? Of what type (e.g., crash, crash-recovery, Byzantine)?
- Time
  - Do I need time assumptions (e.g., upper bound on message and execution times, synchronized clocks)?
- Connectivity
  - All processes are connected?
  - The communication links are reliable? Authenticated?
- Cryptography
  - What cryptography assumptions are needed?
- Architecture
  - Homogeneous or heterogeneous?

Five Distributed Computing Fundamental Results

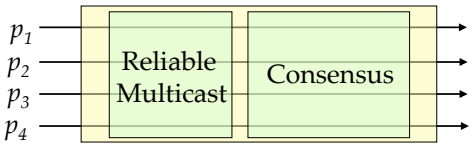
- Impossibility of reliable communication
- Equivalence between total order and consensus
- Impossibility of fault-tolerant consensus
- Minimum synchrony required for FT consensus
- Fault thresholds: *f+1*, *2f+1*, *3f+1* …

Impossibility of Reliable Communication

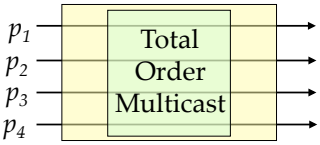
- How can we implement reliable channels on unreliable networks?
  - We can't! We need some weak reliability guarantee in order to build them…
- **Fair channels:**
  - **If a message is sent infinitely many times through a channel, it will be received infinitely by its receiver**
- A practical interpretation:
  - a channel can lose messages for some time
  - eventually, some of these messages will reach the destination
- Reliability can now be implemented:
  - Send a message repeatedly until an ACK is received
  - For BFT, a HMAC should be added to each message, and when it is not valid the message is discarded

Total Order Multicast and Consensus Equivalence

- Total order multicast is equivalent to consensus:
  - A consensus protocol can be used to solve atomic broadcast



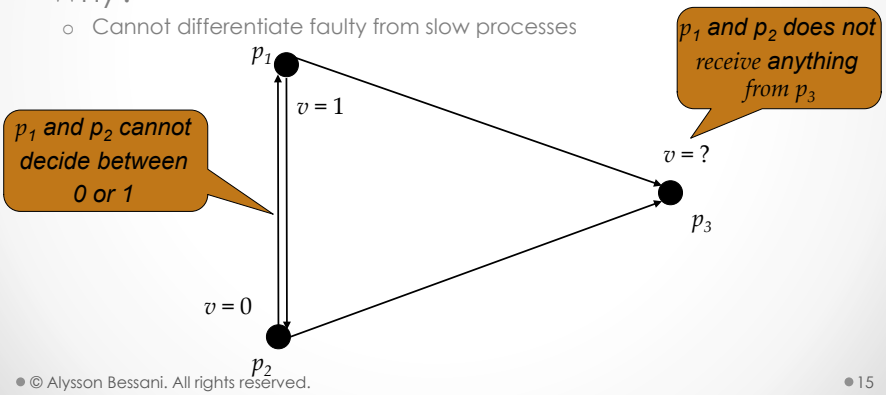
    - **Why it works? every process decide the same set**
  - An atomic broadcast protocol can be used to solve consensus



    - **Why it works? The decision will be the first message delivered first to every process**
- This equivalence holds in most system models

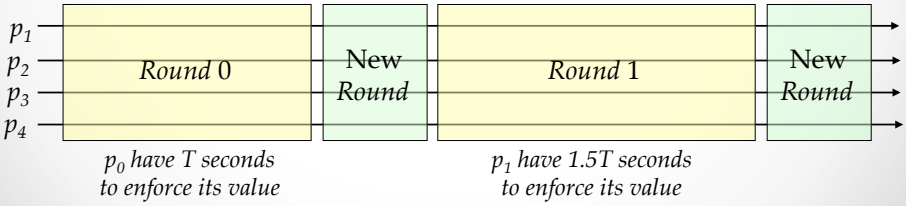
Impossibility of Fault-Tolerant Consensus

- Result:
  Consensus is not solvable in asynchronous systems with reliable channels (or reliable shared memory) even with one crash fault
- Why?
  - Cannot differentiate faulty from slow processes



Minimum Synchrony required for FT Consensus

- Result:
  Fault-tolerant consensus can be solved in the eventually synchronous system model
- Why?
  - The system is asynchronous but has the notion of time
  - After some point, the system will become synchronous (bounded but unknown communication and processing delays)
  - If the algorithm keeps trying (always ensuring safety) and increasing the timeout values, it will be able to solve consensus



Fault Thresholds

- State Machine Replication has two phases
  - Ordering ➔ consensus requirements

	Crash	Byzantine
Synchronous	f+1	3f+1/f+1*
Non-Synchronous	2f+1	3f+1

** using signatures*

  - Execution ➔ voting requirements

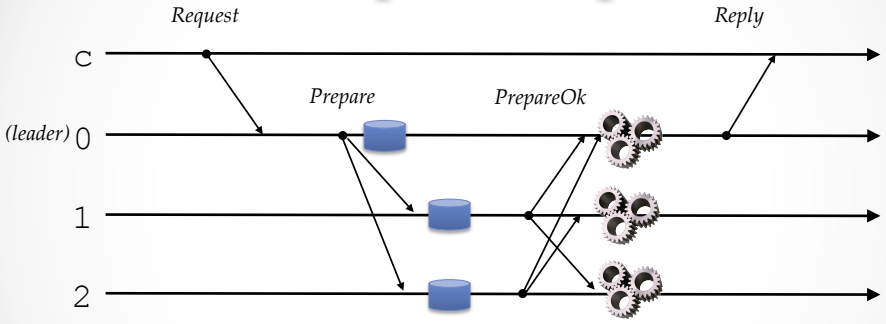
Crash	Byzantine
f+1	2f+1

  - The required number of replicas is the maximum required among these two phases.

(Non-Byzantine) FT State Machine Replication

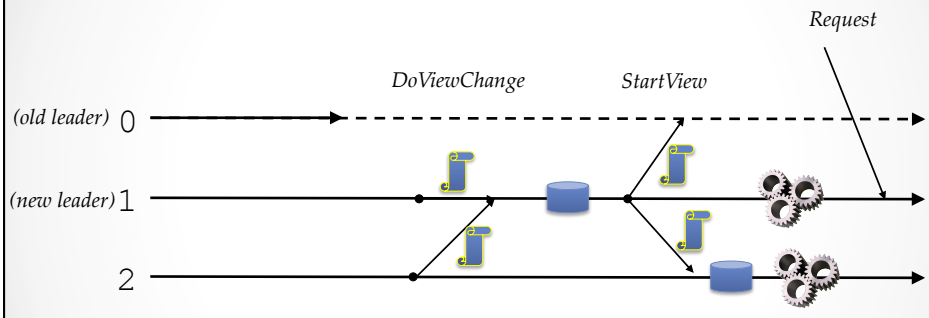
- Paxos (Lamport, TOCS 1998)
  - Agreement framework
  - Can be instantiated as a consensus primitive or a SMR algorithm
  - Three roles: proposer, acceptor and learner
- Viewstamped Replication (Oki & Liskov, PODC'88)
  - Similar to Paxos as a SMR algorithm
  - System model (also similar to Paxos):
    - Unbounded number of crash-prone clients
    - 2f+1 replicas
    - Stable storage
    - Partially synchronous system
      - Safety is always ensured, but Liveness requires synchrony

Viewstamped Replication



- Requests are executed only after the majority of the replicas have it on its log
- It ensures the request will be visible even if the leader fails

Viewstamped Replication



- If a replica suspects the leader, it sends a message to the next leader
- If the next leader receives *f+1* messages, it synchronizes replica logs and start a new view

Some Industrial Applications of Paxos/VR

- Oracle' Berkley DB
  - At least for leader election
- Google' Chubby (Burrows, OSDI'06)
- Google Megastore (Baker et al, CIDR'11)
  - Uses in a different way…
- Yahoo!/Apache Zookeeper (Hunt et al, USENIX'10)
  - Zab is a protocol similar to Paxos
- IBM' Spinnaker (Rao et al, VLDB'11)
- MS' Gaios (Bolosky et al, NSDI'11)
- MS' Windows Azure Storage (Calder et al, SOSP'11)
  - Paxos for intra-datacenter replication

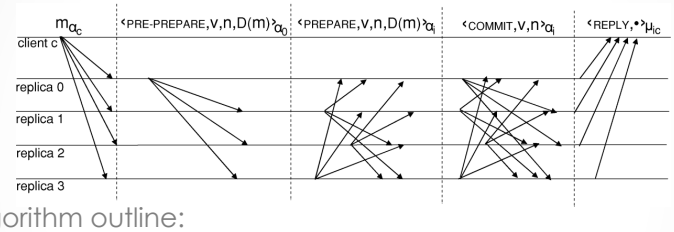
Practical Byzantine Fault Tolerance (PBFT)

- The paper (Castro & Liskov, OSDI'99) sparkled the interest in BFT replication
  - It shows BFT can be fast through the avoidance of public-key crypto (using HMAC vectors instead)
  - Other BFT papers both extend and use the PBFT protocol (and implementation) as a baseline
  - Several versions published: OSDI'99, **TOCS'02**, Liskov' 2010 book chapter

PBFT: System Model

- Asynchronous distributed system
  - Needs partial synchrony for Liveness
- Network can lose, delay, reorder and duplicate messages; but cannot do that indefinitely
  - i.e., they require fair links to implement reliable channels
- Byzantine fault model
  - Fault independence (i.e., no common mode faults)
  - *N = 3f+1* servers, being at most *f* faulty
  - An unbounded number of clients, all of them can be faulty
- Cryptography
  - PK signatures to simplify the protocol presentation
  - MAC (each pair of processes share a key)
  - Digests (hashes)

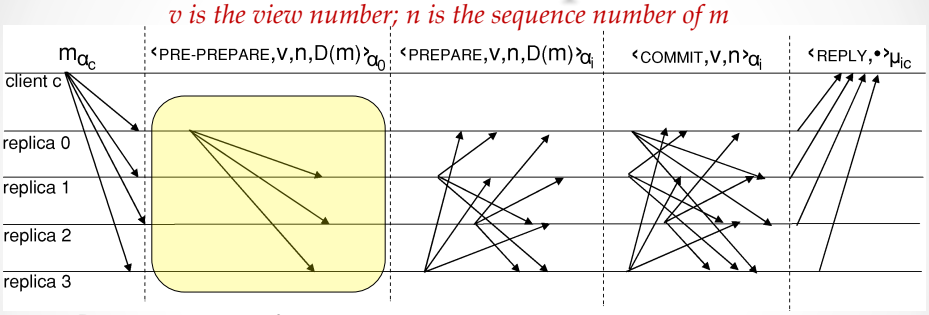
PBFT: Normal Operation



- Algorithm outline:
  - System evolves in views, numbered sequentially. In each view *v*, one server is the primary, the others are the backups: $primary_v = v \bmod N$
  - Client multicasts a signed request to all servers
  - Servers reach agreement about the sequence number of the request
    - The primary proposes the sequence number for each request
    - The backups confirm that the primary follows the protocol
  - If the primary fails, there is a view change
  - Client waits for at least *f+1* replies with the same result (at least one correct server executed the operation and produced the result)

PBFT: Normal Operation I

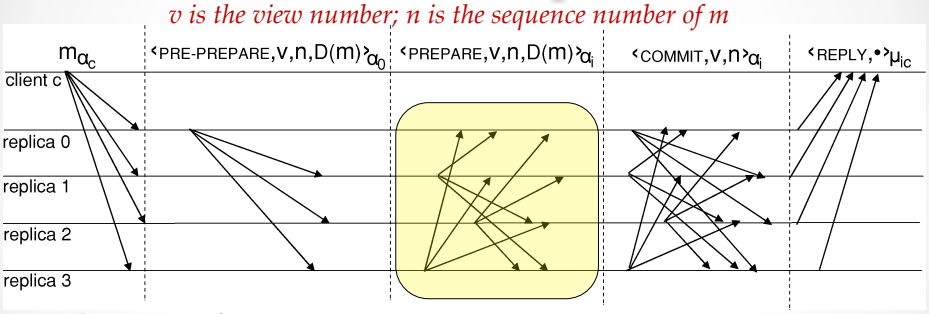
v is the view number; n is the sequence number of m



- Pre-prepare phase:
  - primary receives a correctly signed request *m*
  - It assigns a sequence number *n* to the message and sends this number, a digest of request *D(m)* and its current view number to all backups (other replicas) in a PRE-PREPARE message
  - backup replicas receive the message and test its validity, i.e., if *n* was not assigned to another request and if it is in view *v*
  - If a replica has *m* and a valid PRE-PREPARE for it, it proceeds to the prepare phase (*m is pre-prepared*)

PBFT: Normal Operation II

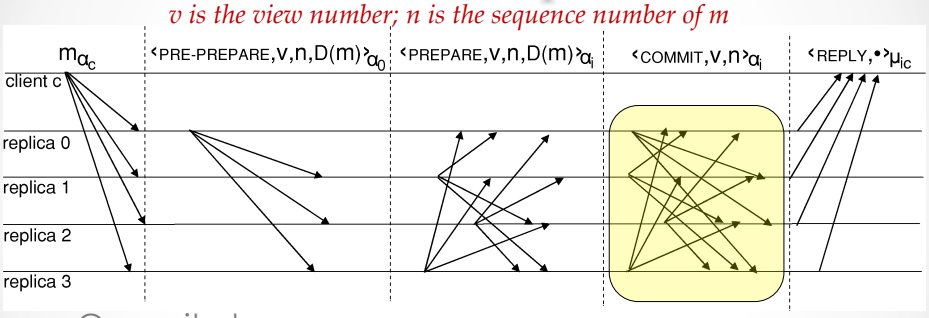
v is the view number; n is the sequence number of m



- Prepare phase:
  - replicas store the received PRE-PREPARE message
  - each replica sends a PREPARE message to other replicas containing *v*, *n* and the digest *D(m)* of the message
  - all servers that receive *2f* PREPARE message from other replicas with the same *v*, *n* and *D(m)*, proceed to the commit phase
  - when a replica finishes the prepare phase for *m*, we say that *m is prepared* on this replica

PBFT: Normal Operation III

v is the view number; n is the sequence number of m



- Commit phase:
  - each replica multicasts a COMMIT message containing *v* and *n*
  - the request *m* for which *n* was assigned is executed when:
    - a replica receives *2f* COMMIT messages with the same *v* and *n* from other replicas
    - all requests with sequence number lower than *n* are executed
  - when the replica *i* finishes the commit phase we say that *m is committed* in *i*

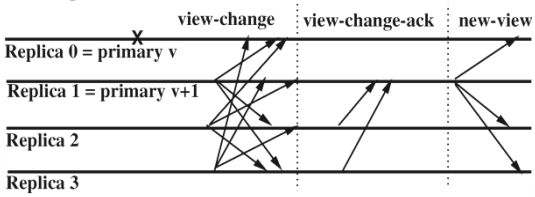
PBFT: Protocol Invariants

- *<m,n,v> is prepared* in a correct replica →
  2f+1 replicas *pre-prepared <m,n,v>* →
  at least *f+1* of them are correct →
  (f+1) + (2f+1) > 3f+1 (any 2f+1 quorum of the system will contain at least one of these correct replicas) →
  it is impossible to have *<m',n,v> prepared (m' ≠ m)* on some correct replica (a correct replica will not pre-prepare two messages with the same *n* and *v*)

- *<m,n,v> is committed* in a correct replica →
  2f+1 replicas *prepared <m,n,v>* →
  at least *f+1* of them are correct →
  any 2f+1 quorum of this system will contain at least one of these correct replicas (that can show that *<m,n,v> is prepared*)

PBFT: Checkpoint and View Change

- Checkpoints
  - All prepared and committed messages are logged in memory
  - Periodically, replicas exchange messages to save a stable checkpoint and truncate the log
- View Change Protocol

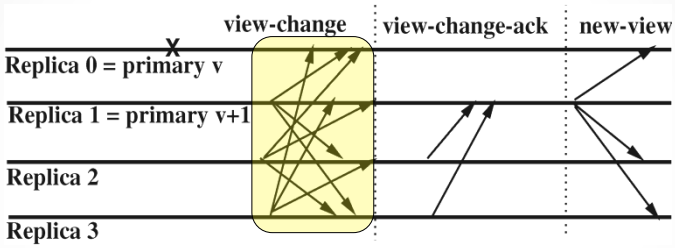


  - If *2f+1* replicas suspect the primary of view *v*, a new view is started
  - The objective of this protocol is to make the correct replicas agree about a new primary and the state of the log

PBFT: Checkpoint

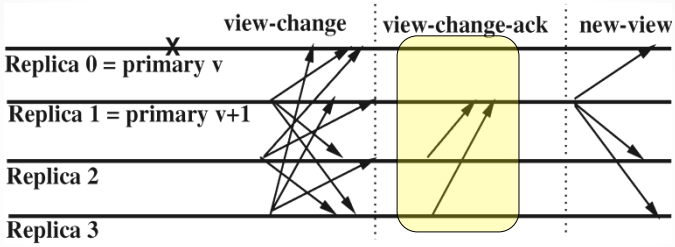
- Every protocol message is only accepted (and logged) if the assigned sequence number falls on a certain interval marked by two values: *h* and *H = h + L* (maximum log size)
- Periodically (every *K* request executions), the replicas exchange CHECKPOINT messages to advance *h* and *H* by *K*
- CHECKPOINT messages contain a digest of system' state before the checkpoint and the sequence number *n* of the last executed request to reach this state (*n mod K = 0*)
- Replicas store *2f+1* CHECKPOINT messages as a proof that no other checkpoint for n is possible
  - *(2f+1) + (2f+1) = 4f+2*; even with *f* Byzantine *4f+2 – f > 3f+1*
- All messages regarding requests with sequence number small than *n* can be discarded from the log
- Late replicas can update themselves fetching states that can be proved correct with *2f+1* CHECKPOINT messages

PBFT: View Change I



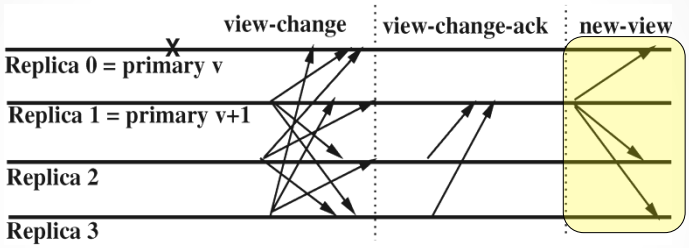
- A backup replica triggers the view change protocol if it stays with some pending message *m* for more than a certain time limit (request timeout expires)
- At this point, the replica stops accepting messages for *v* and sends a VIEW-CHANGE message containing:
  - the next view number *v+1*
  - the sequence number *n* of the last stable checkpoint
  - a set *C* of *2f+1* signed CHECKPOINT messages that validate *n*
  - a set *P* of messages prepared in *i* on views *v' ≤ v*
  - a set *Q* of messages pre-prepared in *i* on views *v' ≤ v*

PBFT: View Change II



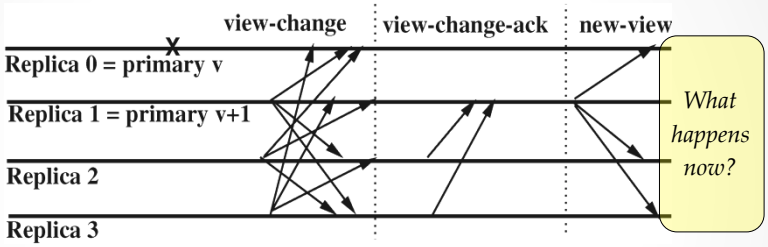
- VIEW-CHANGE messages are accepted if *C* validates *n* and all messages in *P* and *Q* are from views ≤ *v*
- for each accepted VIEW-CHANGE message, a replica sends a VIEW-CHANGE-ACK to the primary of the next view *(v+1)*
- the new primary only accept a VIEW-CHANGE from a replica if it receives *2f-1* VIEW-CHANGE-ACKs for it from other replicas
  (the conference paper you read on assignment 2 does not contain this phase, but it requires PK signatures on view changes)

PBFT: View Change III



- the new primary uses the information on accepted VIEW-CHANGE messages to define new view's *h* as the highest sequence number found on a valid checkpoint
- for each sequence number *n* such that $h < n \leq h + L$
  - if there is some message *m* prepared with *n* in *2f+1* replicas (possibly commited in some of them), the sequence number *n* must be assigned to *m*
  - otherwise, *n* must be assigned to a *null* operation (this only fill gaps)
- these assignments must be sent to other replicas in a NEW-VIEW message together with a digest from each accepted VIEW-CHANGE message used to define them

PBFT: View Change IV



- each backup replica that receive the NEW-VIEW obtains the VIEW-CHANGE messages used to build it
  - they can have it already or they can fetch them from other replicas
- with these messages, each <message, sequence number> assignment contained on the NEW-VIEW message can be verified (with the same procedure used by the primary used to choose these assignments)
  - if there some assignment is invalid, a VIEW-CHANGE for *v+2* is sent to all replicas
  - otherwise, a PREPARE message is sent for each assignment and the protocol resumes to its normal behavior, as if the assignment was a PRE-PREPARE message

Why PBFT works? *(Safety)*

- A Byzantine primary can not "create" its own requests:
  - Backup replicas only process authenticated requests from clients
- A Byzantine primary can not assign the same sequence number to different messages:
  - A correct backup sends a PREPARE message only for the first request it receives for a certain sequence number *n*
  - A correct backup sends a COMMIT message only if it receives PREPARE messages from *2f* other replicas
  - There can not be two different quorums of *2f+1* out-of *3f+1* replicas that send PREPARE messages for the same *n* and different requests
    - These quorums overlap on at least *f+1* replicas
    - Thus, one correct replica should have send contradictory messages, which is not possible.
- Consequently, all replicas execute the same sequence of requests created by clients

Why PBFT works? *(Liveness)*

- A Byzantine primary can decide not to send PRE-PREPARE messages for some requests or to skip sequence numbers:
  - However, when a backup replica receives a request from a client it starts a timer, which is stopped when the request is executed
  - If the timer expires, the backup trigger the view change protocol
  - When enough backup replicas trigger a view change, a new primary is defined and a new view is installed
- For each timer expiration, the timer value is doubled
- Liveness is ensured as long as eventually a timer value suffices to finish the protocol execution with a correct primary

PBFT: Optimizations I

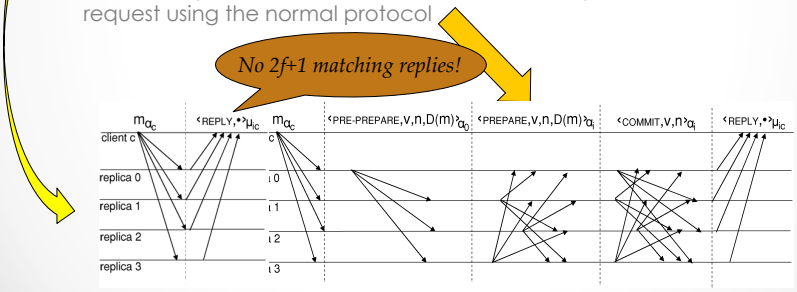
- One of the **key contributions** of PBFT are its optimizations
- Rationale for optimizations:
  "Faults, concurrency and asynchrony are very rare"

PBFT: Optimizations II

- MAC vectors instead of digital signatures
  - The use of PK signatures were the main reason for the poor performance of previous protocols
  - MAC vectors are weaker than digital signatures, so the former cannot always be used to substitute the later
- Digest replies
  - Instead of all replicas sending the reply of a request, the client can choose just one replica to send the reply, the others only send a digest of the reply to allow voting
  - If the received reply is wrong, the client can ask for a (full) reply from other replicas
- Batching
  - Instead of running the agreement protocol for every request to be executed, it can be done for request sets (batches)

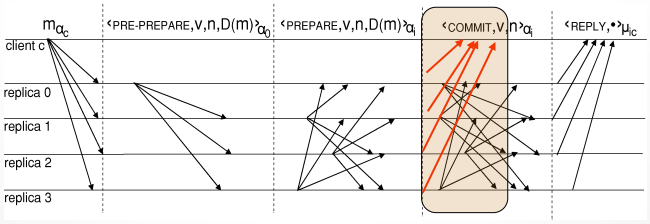
PBFT: Optimizations III

- Read-only requests
  - Read-only requests generally does not require ordering because they do not change the system' state
  - All replicas can immediately reply to the client and it can finishes the read if there are *2f+1* matching replies - instead of *f+1*, to ensure Linearizability
  - Otherwise (due to faulty replicas or concurrency), the client retries the request using the normal protocol



PBFT: Optimizations IV

- Tentative execution
  - Replicas can tentatively execute a request when it is prepared and they have committed all requests with lower sequence number
  - This reduces the protocol latency from 5 to 4 communication steps
  - The client needs to wait for *2f+1* matching replies from different replicas to be sure that the execution order will eventually commit
  - If the client do not receive these replies and a timer expires, it resends the request without asking for tentative execution



References

- Schneider. Implementing Fault-Tolerant Services using the State Machine Approach: a Tutorial. ACM Computing Surveys 1990.
- Lamport. The Part-time Parliament. ACM TOCS 1998
- Oki & Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. PODC'88
- Burrows. The Chubby Lock Service for loosely-coupled distributed systems. OSDI'06
- Baker et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. CIDR'11
- Hunt et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. USENIX'10

References

- Bolosky et al. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. NSDI'11
- Rao et al. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. VLDB'11
- Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. SOSP'11
- Castro & Liskov. Practical Byzantine Fault Tolerance. OSDI'99
- Castro & Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. ACM TOCS 2002
- Liskov. From Viewstamped Replication to Byzantine Fault Tolerance. Replication: Theory and Practice, 2010

Part II

BFT Literature Review

EuroSys 2012

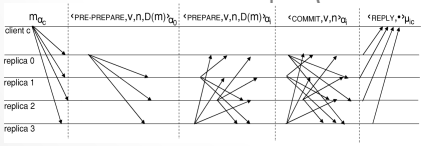
Outline

- Improving BFT performance
- Robust BFT protocols
- Architectural hybridization
- Implementation techniques
- Complementary techniques for BFT

Note: *there are other papers and other aspects, but this is my selection given the time constraints we have*

Improving BFT Performance

- PBFT performance is competitive with crash fault-tolerant systems, and in some cases even with non-replicated systems
- However, in the expected common situation where
  - There are no faults
  - The system is synchronous
  - There is no concurrency
- PBFT still requires $2(n-1)^2+(n-1)$ messages and 5 communication steps (without optimizations)

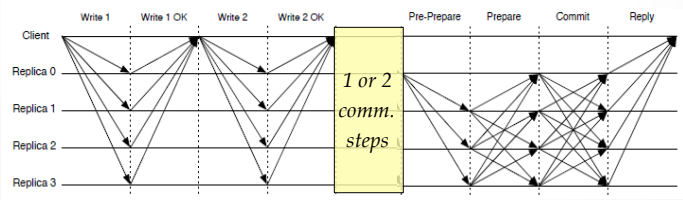


Improving BFT Performance

- Since PBFT publication, several works tried to improve its performance
- Q/U – Query/Update (Abd-El-Malek et al, SOSP'05)
  - "Pure" quorum-based protocol that works on asynchronous system
  - Advantages:
    - Improves the fault scalability of the system, i.e., the throughput of the system does not drop dramatically when *f* increases
    - Operations require only two communication steps (best case)
  - Drawbacks:
    - Sacrifices Liveness (Obstruction-freedom instead of Wait-freedom): operations only terminate if there is no write contention on the object
    - Requires $n \geq 5f+1$

Improving BFT Performance

- HQ-Replication (Cowling et al, OSDI'06)
  - Combines quorum-based protocols with PBFT
    - If there is no concurrency, executes a (f-dissemination BQS) write protocol to change the system state
    - If concurrency is detected, start PBFT to order concurrent requests
  - Same advantages of Q/U, with the same Liveness guarantees of PBFT and using only *3f+1* replicas

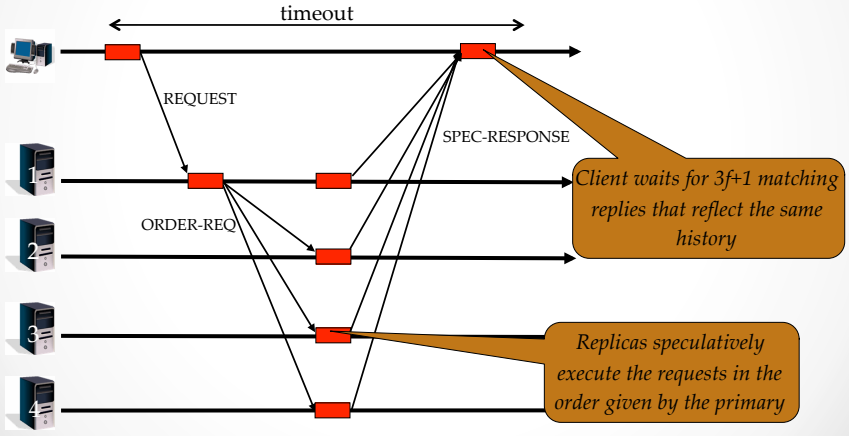


Zyzzyva: Speculative BFT *(Kotla et al, TOCS 2009)*

- The "final word" on high-performance BFT protocols
- Main idea: PBFT with speculative execution
  - Each replica (speculatively) executes a request just after receiving the sequence number of this request by the primary
  - After executing the request they send a reply to the client
  - The consistent state of the replicas only matter to clients, so let them verify if all replicas are on the same state
  - If there is some problem (e.g., the primary sends different operations to different replicas), a correct client will detect it
  - This client will inform the replicas, which must rollback to a safe state and change the primary
- Improves latency and throughput on the best case
  - Zyzzyva requires only 3 communication steps
  - Zyzzyva requires only 2*n* message exchanges

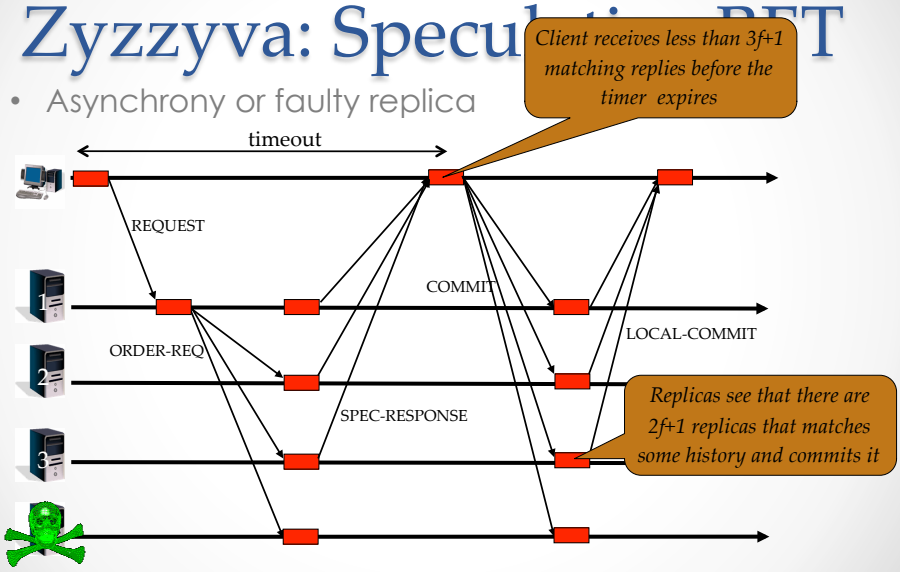
Zyzzyva: Speculative BFT

- Best-case execution (synchronous and fault-free)



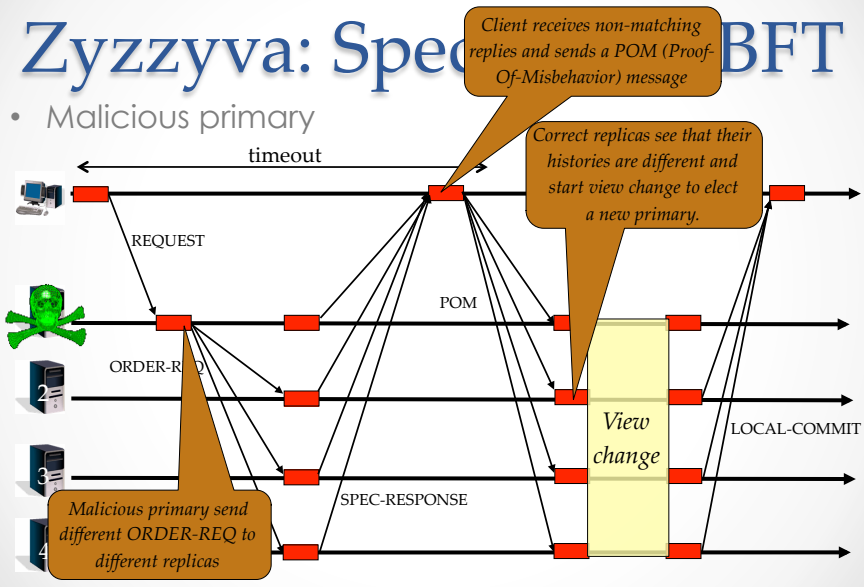
Zyzzyva: Speculative BFT

- Asynchrony or faulty replica



Zyzzyva: Spec BFT

- Malicious primary



Zyzzyva: Speculative BFT

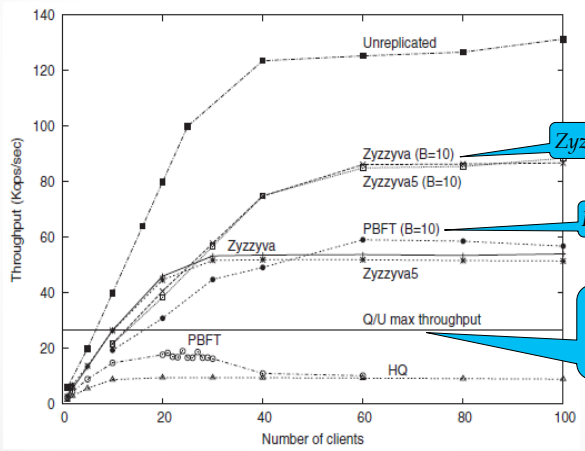
- Comparison with other protocols (theory)

		PBFT	Q/U	HQ	Zyzzyva	State Machine Repl. Lower Bound
Cost	Total replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	3f+1 [Pease et al. 1980]
	App. replicas	**2f+1**	5f+1	3f+1	**2f+1**	2f+1 [Schneider 1990]
Throughput	MAC ops/request	2+(8f+1)/b	2+8f	4+4f	**2+3f/b**	2†
Latency	NW 1-way latencies	4	**2***	4	**3**	2* or 3‡

These systems tolerate f faults using MACs for authentication [Castro and Listov 2002] and use a batch size of b [Castro and Listov 2002]. Bold entries denote protocols that match known lower bounds or those with the lowest known cost. †It is not clear that this trivial lower bound is achievable. ‡The distributed systems literature typically considers 3 one-way latencies to be the lower bound for agreement on client requests [Dutta et al. 2005; Lamport 2003; Martin and Alvisi 2006]; *A delay of 2 one-way latencies is achievable if no concurrency is assumed.

Zyzzyva: Speculative BFT

- Comparison with other protocols (experimental)



- n = 4
- no faults
- 0 byte requests
- *null* operations

Zyzzyva (batch): 84Kops/s

PBFT (batch): 60Kops/s

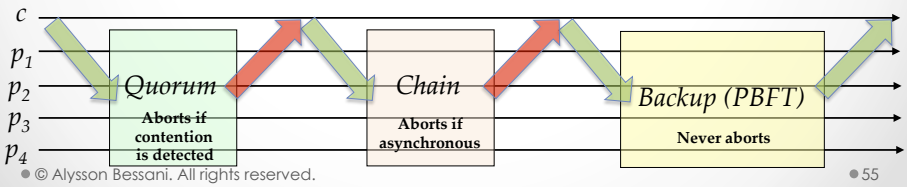
Q/U and HQ: 23Kops/s (quorum-based protocol cannot batch messages)

Zyzzyva: Speculative BFT

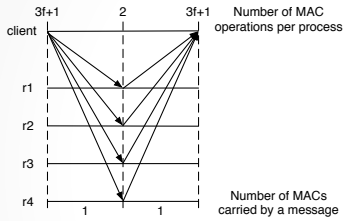
- Zyzzyva is the fastest protocol one can devise for ordering requests under the Byzantine fault model
- However, it is not perfect
  - Speculative execution on servers might not be a good idea
    - You need to be able to rollback to a committed state if a view change is triggered
    - This makes your server code much more complicated
  - If you wait for replies from all replicas, you will always be waiting for the slower one
    - In non-synchronous networks you will have to calibrate your timeout value carefully
  - Zyzzyva is vulnerable to several attacks, just like PBFT

The Next 700 BFT Protocols

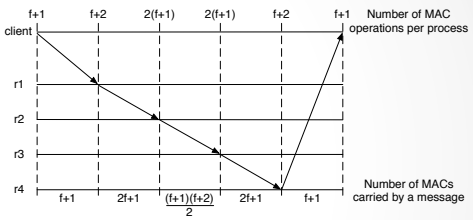
- HQ and Zyzzyva are protocols with fast and slow paths, being the slow path similar to PBFT
- Guerraoui et al (EuroSys'10) generalized this idea with the ABSTRACT abstraction
  - An ABSTRACT instance is just like a state machine replication, but abortale
  - ABSTRACT instances are composable, i.e., if one instance aborts, it returns enough information for clients to start another
  - This allows the development of optimistic protocols that can revert to more conservative approaches if the expected conditions are not meet
- *Aliph BFT SMR protocol:*



The Next 700 BFT Protocols



Quorum (latency-optimal)



Chain (throughput-optimal)

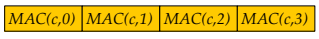
ABSTRACT is a nice idea that really simplifies the design of optimistic state machine replication.

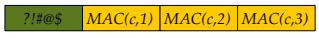
Robust BFT Protocols

- All distributed protocols can have their performance hurt by (Distributed) DoS attacks
  - There is nothing we can do about that… we need communication and timing assumptions in order to solve BFT consensus
- However, the quest for optimizing these protocols for the "*expected common case*" made them even more fragile to malicious behavior
  - E.g., malicious clients can try to execute operations continuously on systems like HQ and Q/U to make their operation extremely slow
- However, there are two attacks (≠ (D)DoS) that can really hurt the performance of systems like PBFT and Zyzzyva (Amir et al., DSN' 08, TDSC 2011)

 57

BFT Under Attack

- Attack #1: causing view change with a malicious client without using DoS
  - On PBFT, clients need to send a request "signed" with an authenticator (a MAC vector)
    - Correct authenticator: 

  - A malicious client can send a corrupted authenticator that is valid for all backup replicas but not for the primary
    - Malicious authenticator: 

    - The primary will ignore the client's request
    - Other replicas will accept it and, after their timer expires, will relay it to the primary
  - Since the primary will never accept this request, other replicas will start a view change after a second timeout
  - Conclusion: the use of authenticators allow faulty clients to force view changes as they wish

 58

BFT Under Attack

- How to "patch" attack #1' vulnerability?
  - Make clients sign (not with MAC vectors) their messages
  - Digital signatures (like RSA) ensure that *if some correct server authenticate the message, then all correct servers will authenticate the message*
  - Performance issues:
    - Clients generate signatures, servers only verify one signature per request
    - Operation' latency increases by a signature (~5 ms on standard hardware) plus a verification (~0.5 ms)
    - Throughput becomes limited by the amount of signatures each server can verify per second
      - The mentioned single core machine cannot do more than 2Kops/s
      - But a 4-year-old quad-core machine can do ~40Kops/s

BFT Under Attack

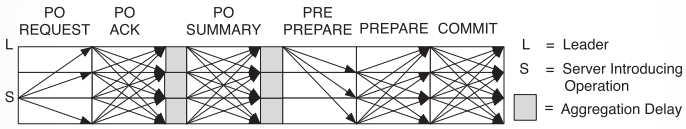
- Attack #2: degrading performance with a faulty primary
  - A faulty primary must order a request before other replicas timer expires for this request
  - Assuming $T_{timeout}$ = 100 ms, if a faulty primary delays the ordering of each request by 90 ms, a view-change will not be triggered
  - Nonetheless, the performance of the system will drop dramatically
  - This attack can be even more devastating if combined with attack #1, since for each view change $T_{timeout}$ is doubled
  - Conclusion: a faulty primary can inject a delay of almost $T_{timeout}$ ms on each request processing, making the end-to-end performance of the system orders of magnitude worse than expected

BFT Under Attack

- How to mitigate attack #2?
  - Solution 1: use decentralized (leader-free) protocols
    - The request' sequence number is not defined by a primary
    - Replicas will propose their set of pending requests for ordering in a decentralized consensus (Moniz et al, TDSC 2011)
    - Whether or not this approach works depends on how similar the proposals are, i.e., if all replicas receive client's requests on the same order (which happens very often on HUB-based networks)
  - Solution 2: monitor the primary's performance and start a view change if it's too low
    - Problem is *"how to define the threshold between an unstable network and a faulty primary"*
    - A wrong view change can hurt the protocol' performance
  - Solution 3: rotate the primary periodically

Protocols Solving these Issues

- Prime (Amir et al, DSN'08, TDSC 2011)
  - Identified these problems for the first time in their DSN'08 paper
  - The Prime protocol adds a pre-order phase to PBFT

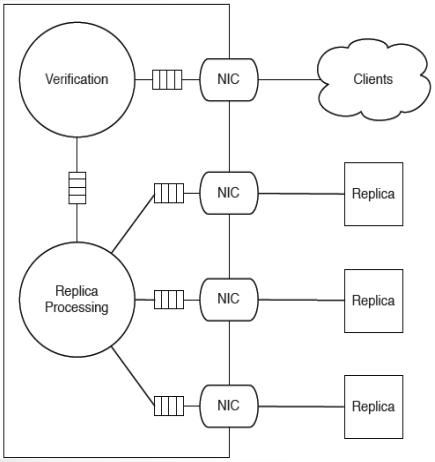


- Aardvark (Clement et al, NSDI'09)
  - PBFT made robust
- Spinning (Veronese et al, SRDS'09)
  - Rotating-coordinator BFT SMR

Robust BFT SMR

- Clement et al. (NSDI'09) proposes a variation of PBFT that implements robust state machine replication
  - The name of the protocol is ***Aardvark*** ☺
- By robust, it means:
  - Maintains a stable performance even when under attack by *f* malicious replicas and an unbounded number of clients
- Three main differences (when compared with PBFT):
  - Clients must sign requests
    - to avoid malicious clients provoking view changes
  - Resource Isolation
    - to resist denial of service attacks against network interfaces
  - Regular view changes
    - to avoid performance degradation attacks

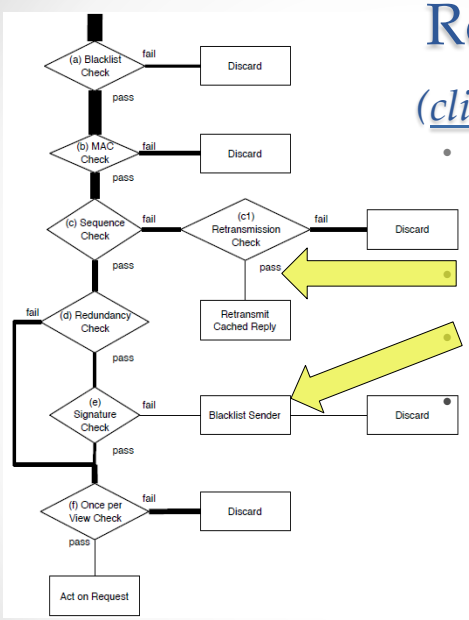
Robust BFT SMR: Replica Architecture



- Every replica needs *n* NICs (one to each other replica plus one to clients)
- This makes the system resilient to DoS network attacks from faulty replicas and clients
- To help resist DoS attacks, there are specific algorithms to verify client requests and process replica messages

Robust BFT SMR

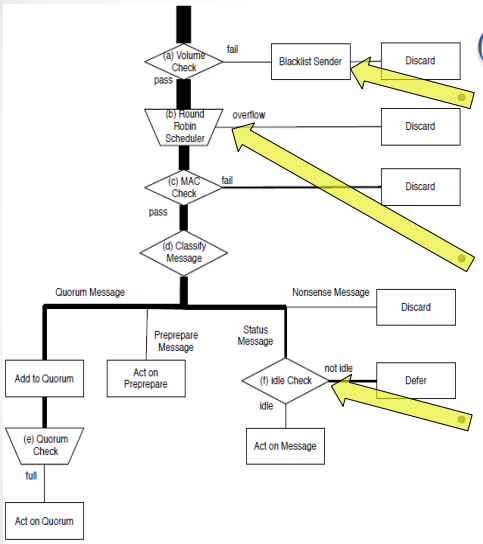
(client request verification)



- Client messages have both a MAC and a signature
  - Why?
- Each reply is cached to deal with retransmissions
- Clients that misbehave are blacklisted
- "Redundancy" and "once per view checks" take care of replay attacks
  - Clients need to sign different requests to make them valid

Robust BFT SMR

(replica msg processing)



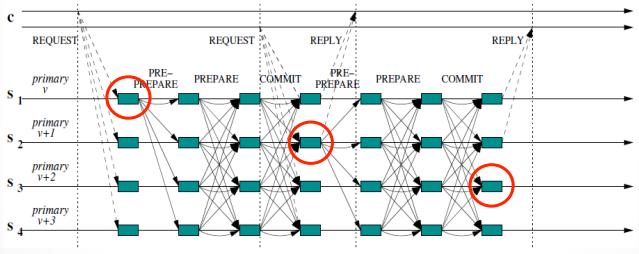
- If some replica is sending 20 times more messages than the others, blacklist it
- To avoid resource exhaustion, messages are processed on a round robin fashion
- Only process catch up messages if the system is idle

Spinning

- A protocol build upon PBFT, but with a modification based on a simple idea:
  - PBFT's problem is that a malicious primary can keep ordering requests very slowly without triggering view changes
  - So, why not change view after each message commit?
  - in this way, the sequence number of a message matches exactly the view number of its delivery
- Potential problem:
  - The view change protocol is complex and costly
  - But it is not a problem: the view change will deterministically happen after every committed message, so it is not necessary to have a special protocol to change primary
- The resulting protocol (Spinning) makes the primary role rotates to all servers

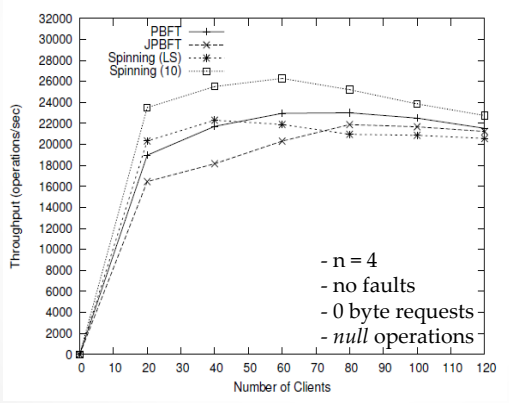
Spinning

- Example execution of Spinning:
  - first request is ordered by s_1, which is the primary of view *v*
  - second request is ordered by s_2, which is the primary of *v+1*
  - …



Spinning: Performance

- Changing primary improves or degrades performance in fault-free executions?

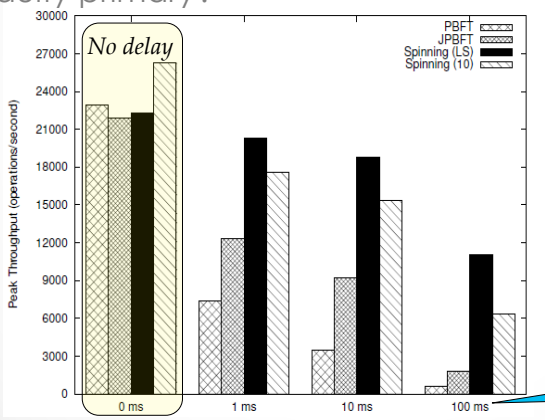


No, the performance is better!

The primary extra load is evenly distributed between all replicas

Spinning: Performance

- What happens when a latency is injected by a faulty primary?



Spinning performance degrades much slower than PBFT

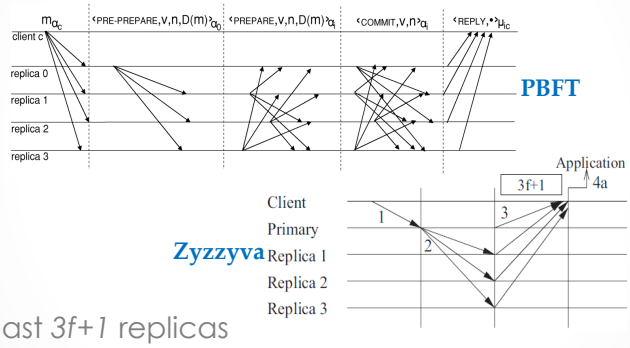
Malicious primaries can only degrade the performance of the system in *f* out-of *n* protocol executions

Spinning: Issues

- Without the repair procedure of view changes, how replicas recover from a malicious primary on some view?
  - Merge operation: joins one or more faulty views (i.e., with faulty primaries) with a correct view (i.e., with correct primary)
  - The idea is very similar to PBFT's view change: the new correct primary will read the state of the system and proceed ordering requests ensuring the protocol invariants
- Faulty replicas can force merge operations periodically
  - To avoid that, after a merge operation, the primary of the most recent merged view is put on a blacklist
  - Every time a replica goes to a black list, it stays there for a number of turns (*n* views) equal to the number of times it was blacklisted
    - First time, loses 1 turn; second time, loses 2 turns, and so on…
  - Blacklisted replicas cannot be primary on their views, but otherwise can participate on the protocol as a backup

Architectural Hybridization

- **Motivation: BFT in Homogeneous Systems is Expensive**



- At least *3f+1* replicas
- At least *3* communication steps for establish agreement (non-speculative normal case operation)

Architectural Hybridization

- Is it possible to do better?
  1- Less than *3f+1* replicas to tolerate *f* Byzantine faults?
  - *Homogeneous non-synchronous systems require 3f+1 replicas*

  (and, at the same time)
  2- Less than 3 communication steps to establish agreement
  - *It is possible to solve consensus with 2 communication steps if there are 5f+1 replicas* (Martin & Alvisi, TDSC 2007)

- Hybrid distributed systems (Veríssimo, SIGACT News 2006) with **local trusted components** can do that!

History: Trusted Components and BFT

- (Correia et al, SRDS'02) BFT Reliable Multicast using TTCB, a **distributed real-time trusted component**
- (Correia et al, SRDS'04) BFT SMR with *2f+1* replicas using a **distributed trusted component**
- (Chun et al, SOSP'07) PBFT with *2f+1* replicas using a **complex local trusted component** (A2M)
- (Levin et al, NSDI'09) A2M reduced to a **simple secure counter** (TrInc), that **can be build using a TPM chip**
- (Veronese et al, DI-FCUL TR 2008, TC 2011) MinBFT shows that with a **trusted counter** one can **reduces BFT SMR to viewstamped replication/Paxos**
- (Kapitza et al, EuroSys'12) BFT SMR with only ***f+1* active replicas** using a **trusted counter efficiently implemented in FPGA**

MinBFT: System Model

- Eventually synchronous system
- Authenticated and reliable channels
- Local Trusted Component (can only crash)
- Secure hash function
- $n \geq 2f+1$ replicas, at most f can suffer Byzantine faults

 75

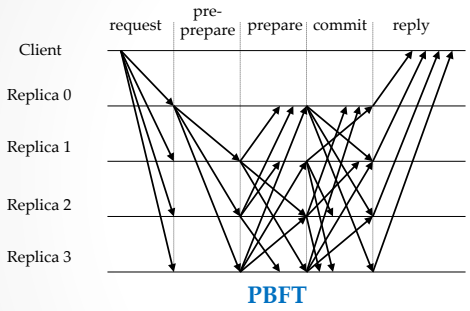
MinBFT Trusted Component

(USIG - Unique Sequential Identifier Generator)

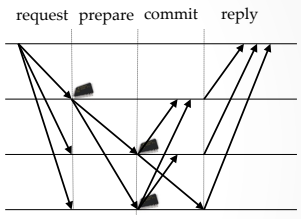
- A minimal local trusted component containing
  - A cryptosystem for authenticating its outputs
  - A monotonic counter
- Storage (on the USIG of process *i*):
  - A private-key PrK_i
  - An unbounded counter *count*
- Operations:
  - *createUI(m)*
    - Assigns a counter value c_*val* to a message *m*
    - Increments the counter: *count*++
    - Outputs *UI* = <c_val, i, H(m), $Sign_{PrKi}$>
  - *verifyUI(j, PuK_j, UI, m)*
    - Verifies if UI was generated by createUI(m) on the USIG of process *j*. This verification uses *j*'s USIG public key PuK_j
    - Outputs *true* or *false*

 76

PBFT x MinBFT



PBFT



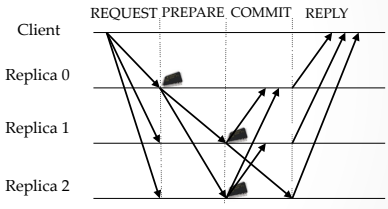
MinBFT

Benefits of MinBFT

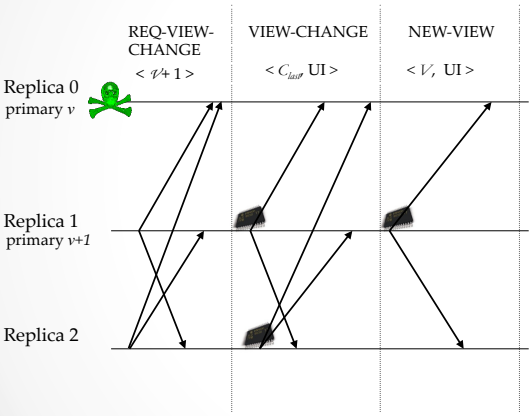
- *2f+1* instead of *3f+1* replicas (minimal for general SMR)
- 2 steps instead of 3 on the normal case (minimal for consensus)
- USIG is arguably a minimal trusted component

MinBFT: Normal Case

- Primary defines the order
  - The sequence number is the USIG counter value assigned to the message
- Servers accepted *f+1* commits
  - Each one should have a valid UI from its sender USIG
- Execution follows the order on PREPARE' UI
- Client waits for *f+1* matching replies



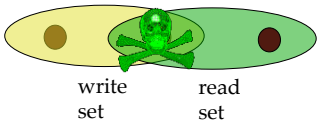
MinBFT: View Change



- When a request is received, a timer T_{exec} is started
- *f+1* REQ-VIEW-CHANGE
- *f+1* VIEW-CHANGE

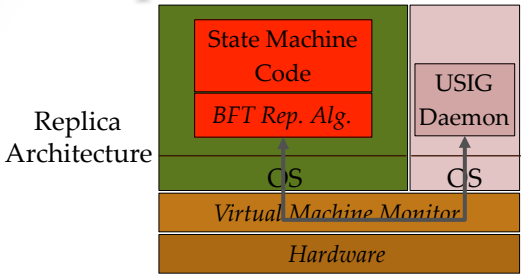
MinBFT: Why it Works?

- Uses *2f+1* replicas with quorums of size *f+1*
- One replica in the intersection of any two quorums



- What if this replica is faulty?
  - It cannot lie because every value is associated with a UI
  - Different messages will have different UIs
- Practical effects:
  - A primary replica cannot send two PREPARE messages with different messages and the same sequence number (UI)
  - A backup replica cannot lie about the value proposed by the primary

USIG Implementation: VM

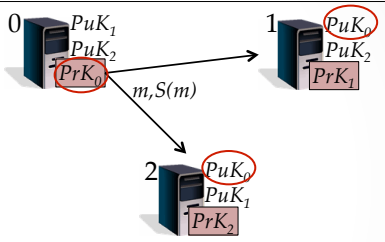


- The BFT protocol + application runs on a untrusted virtual machine that have access to the outside network
- The USIG is implemented as a daemon on a trusted virtual machine (e.g., Xen's *Dom0*)
- They communicate by TCP sockets

USIG Implementation: VM

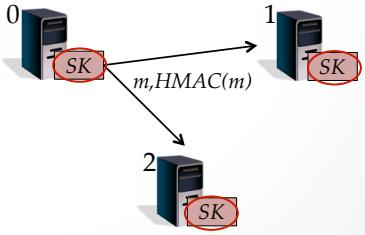
Public-key USIG (RSA or ESIGN)

- Only *createUI* requires trusted component access



HMAC USIG

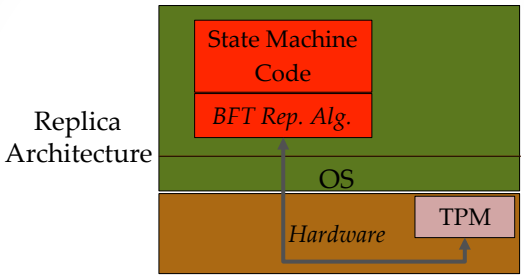
- Both *createUI* and *verifyUI* requires trusted component access



USIG Implementation: VM

- MinBFT can be implemented with both variants, but the HMAC version, albeit potentially more efficient, can be more difficult to manage
  - Symmetric keys have a short life-cycle than PK keys
  - How to refresh them without interrupting the protocol?
- MinZyzzyva requires clients to verify UIs, meaning that clients need to have a trusted component with the shared secret key
  - This is infeasible in practice
  - Conclusion: MInZyzzyva only works with PK USIG

USIG Implementation: TPM



- A public-key (2048-bit RSA) implementation of the USIG service
- The private key and the counter are stored in the TPM (version 1.2 or higher)
- BFT protocol access a TPM driver to issue commands

USIG Implementation: TPM

createUI(m), as called by replica *i*

- First, calculate $h_m = H(m)$
- Send the following commands to the TPM (details omitted)
  1. `TPM_EstablishTransport`
  2. `TPM_ExecuteTransport(TPM_IncrementCounter)`
  3. `TPM_ReleaseTransportSigned(`h_m`)`
- The last command returns:
  - A 2048-bit RSA signature *S* of <`TPM_IncrementCounter`; *c_val*; h_m>
- This UI value is <<`TPM_IncrementCounter`; *c_val*; h_m>, *S*>

Proves that the counter was incremented

Association of a counter value with the message

Proves that it was generated by TPM on replica i

USIG Implementation: TPM

verifyUI(j, PK_j, UI, m)

- Verify the format of the data structure (e.g., there is an increment on the TPM counter)
- Verify if $h_m = H(m)$ is on the UI
- Verify the signature using the public key PK_j
- If all these checks are passed, return *true*, otherwise, return *false*

Important Remark: This function don't need to access the TPM

USIG Performance: VM x TPM

- TPM USIG
  - Signature: 797 ms
  - One increment by 3.5 seconds
  - 32-bit monotonic counter
- VM USIG

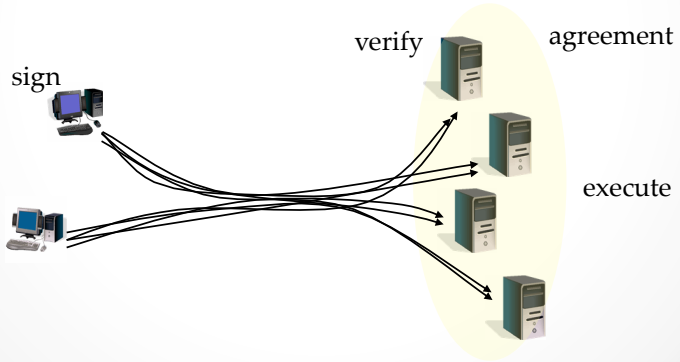
Algorithm	createUI	verifyUI
Hmac (SHA1)	0.008	0.007
ESIGN (2048 bits)	1.035	0.548
RSA (1024 bits)	10.683	0.580

Implementation Techniques

- BASE (Castro et al, TOCS 2003)
  - Define useful abstractions to implement diverse BFT services
- Parallel execution of requests (Kotla & Dahlin, DSN'04)
  - Some service requests do not require total order execution (writes on different files of a file system), and can be executed in parallel
  - May improves the throughput of certain services (e.g., distributed FS)
- On-Demand Replica Consistency (Distler & Kapitza, EuroSys'11)
  - Breaks the service state in partitions
  - Each partition executes a subset of the submitted requests
  - Specially useful if executing a request require a lot of resources
- Separating Agreement from Execution (Yin et al, SOSP'03)
- UpRight (Clement et al, SOSP'09)
- ZZ (Wood et al, EuroSys'11)

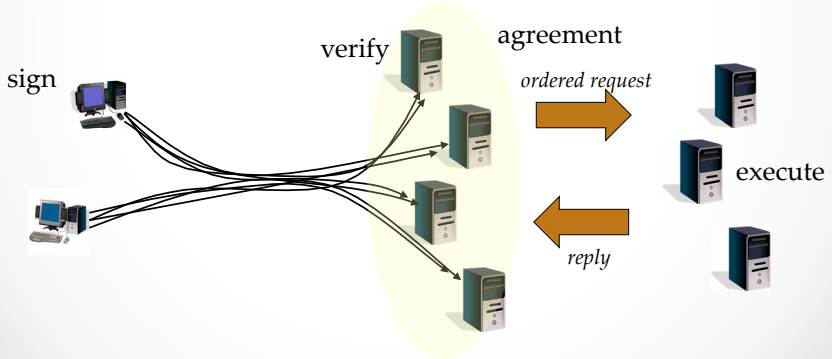
Classical BFT SMR Architecture

- Clients sign requests and sent them to the replicas
- Replicas verify client signature and run an agreement protocol to establish total order
- Replicas execute the request and send the reply to the client



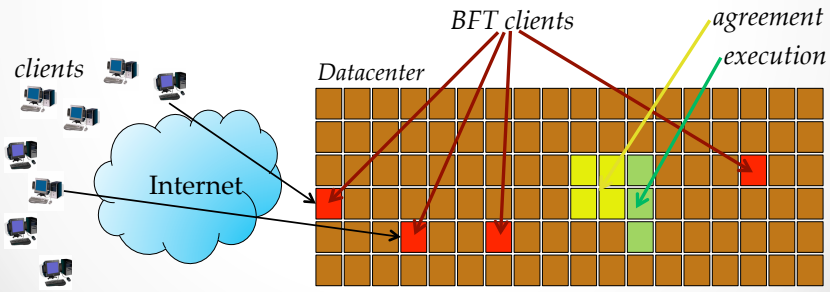
Separating Agreement/ Execution Architecture

- Separate servers in two layers: agreement and execution
- Clients sign requests, agreement replicas verify it
- *3f+1* replicas to agree on requests sequence number and *2f+1* for executing the requests



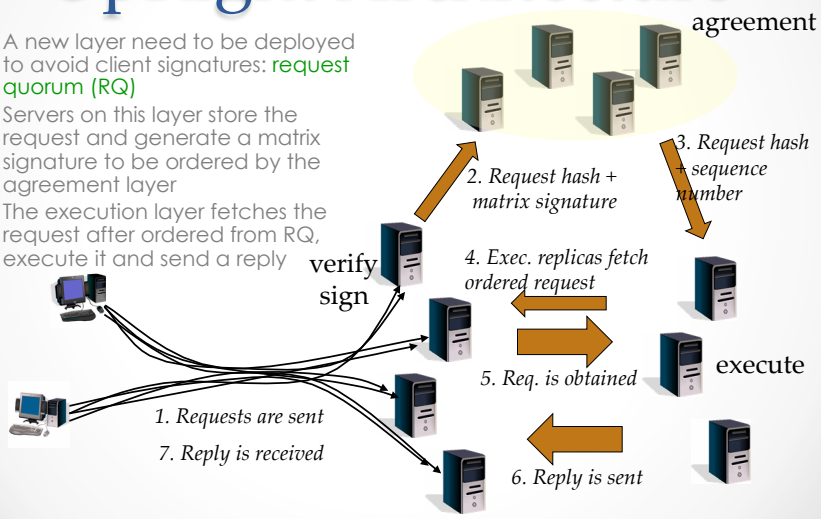
Agreement/Exectuion Problem

- In data centers, clients usually are also servers… they have to be fast (generating signatures is very costly)
  - E.g.: web services (BFT clients) access a BFT database (execution)
- These web service hosts need to serve lots of clients (high throughput) and they are paid by the service provider



UpRight Architecture

- A new layer need to be deployed to avoid client signatures: request quorum (RQ)
- Servers on this layer store the request and generate a matrix signature to be ordered by the agreement layer
- The execution layer fetches the request after ordered from RQ, execute it and send a reply

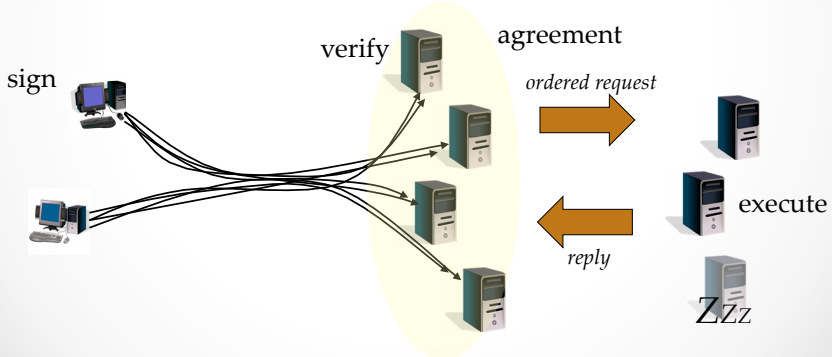


UpRight Remarks

- Number of faults tolerated:
  - Request quorum: $n_r \geq 2u + r + 1$
  - Ordering: $n_o \geq 2u + r + 1$
  - Execution: $n_e \geq u + r + 1$
- Clients only do MACs, not signatures… it is more aligned with cloud applications (clients are also servers of application services)
- Speculative execution is not employed in the service, but only on order assignment (execution servers are just like clients receiving replies from Zyzzyva)

ZZ Architecture

- Key observation: In fault free executions, *f+1* execution replicas are enough for the execution layer
- In server consolidation scenarios, these extra *f* execution replicas can be dormant VMs



Complementary Techniques for BFT: Fault Recovery

- Problem with tolerating *f* faults:
  - If an intelligent adversary is able to compromises *f* machines, given enough time, he/she will compromise *f+1* (or more)
  - Solution: Proactive Recovery (Castro & Liskov, TOCS 2002)
    - Replicas (compromised or not) are cleaned periodically
- PR requires a local trusted real-time component
  - Otherwise, it may be vulnerable to certain attacks (Sousa et al, DSN'05)
  - Most proactive recovery systems are vulnerable (Sousa et al, HotDep'06)
- To ensure availability you may also need *2k* extra replicas if at most *k* recover at the same time

Complementary Techniques for BFT: Diversity

- *f*-fault-tolerant replicated systems are useful only if faults are not correlated
- It usually requires **diverse replicas**
  - Different administrative domains
  - N-version programming (effective?)
  - Obfuscation, Memory randomization (effective?)
  - Use of different components like databases (Gashi et al, TDSC 2007), file systems (Castro et al, TOCS 2003) and operating systems (Garcia et al, DSN'11) is effective!
- *What about deploying and managing diversity?*

References

- Abd-El-Malek et al. *Fault-scalable Byzantine Fault-tolerant Services*. SOSP'05
- Cowling et al. *HQ-Replication: a Hybrid Quorum Protocol for Byzantine Fault Tolerance*. OSDI'06
- Kotla et al. *Zyzzyva: Speculative Byzantine Fault Tolerance*. ACM TOCS 2009 (prel. SOSP'07)
- Guerraoui et al. The Next 700 BFT Protocols. EuroSys'10
- Amir et al. *Byzantine protocols Under Attack*. IEEE TDSC 2011 (prel. DSN'08)
- Moniz et al. RITAS: Services for Randomized Intrusion Tolerance. IEEE TDSC 2011
- Veronese et al. *Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary*. SRDS'09

References

- Clement et al. *Making Byzantine Fault-tolerant Systems Tolerate Byzantine faults*. NSDI'09
- Martin & Alvisi. Fast Byzantine Paxos. IEEE TDSC 2007
- Veríssimo. Travelling through Wormholes: a new look at Distributed System Models. SIGACT News 2006
- Correia et al. *Hybrid Byzantine-resilient Reliable Multicast*. SRDS'02
- Correia et al. *How to Tolerate Half less One Byzantine Faults in Practical Distributed Systems*. SRDS'04
- Chun et al. Attested append-only memory: Making adversaries stick to their word. SOSP'07
- Levin et al. TrInc: Small Trusted Hardware for Large Distributed Systems. NSDI'09

References

- Veronese et al. Efficient Byzantine Fault Tolerance. IEEE TC 2011. to appear (prel . DI-FCUL Tech. Report 2008)
- Kapitza et al. ChepBFT: Resource-efficient Byzantine Fault Tolerance.EuroSys'12
- Castro et al. BASE: Using Abstractions to Improve Fault Tolerance. ACM TOCS 2003
- Kotla & Dahlin. High-throughput Byzantine Fault Tolerance. DSN'04
- Distler & Kapitza. *Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. EuroSys'11*
- Yin et al. *Separating Agreement from Execution in Byzantine Fault-tolerant Services*. SOSP'03

References

- Clement et al. UpRight Cluster Services. SOSP'09
- Wood et al. ZZ and the Art of BFT Execution. EuroSys'11
- Sousa et al. How resilient are distributed f fault/ intrusion-tolerant systems? DSN'05
- Sousa et al. Hidden Problems of Asynchronous Proactive Recovery. HotDep'07
- Gashi et al. Fault tolerance via diversity for off-the-shelf products: a study with SQL database servers. IEEE TDSC 2007
- Garcia et al. OS Diversity for Intrusion tolerance: Myth or Reality? DSN'11

Other Aspects

Wide-area replication

- Wester et al. Tolerating Latency in Replicated State Machines Through Client Speculation. NSDI'09
- Mao et al. Towards Low Latency State Machine Replication for Uncivil Wide-area Networks. HotDep'09
- Amir et al. STEWARD: Scaling Byzantine Fault-Tolerant Replication to Wide-Area Networks. IEEE TDSC 2010
- Veronese et al. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. HASE'10

Weak consistency & others

- Li & Mazières. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. NSDI'07
- Singh et al. Zeno: Eventually Consistent Byzantine-Fault Tolerance. NSDI'09
- Sen et al. Prophecy: Using History for High-Throughput Fault Tolerance. NSDI'10
- Bessani et al. Active Quorum Systems. HotDep'10

Part III

• • •

Applications, Open Problems & Practice

EuroSys 2012

BFT Applications

- Distributed File Systems
  - BFS (Castro & Liskov, TOCS 2002), BASEFS (Castro et al, TOCS 2003)
  - Oceanstore (Kubiatowicz et al, ASPLOS'00), Farsite (Adya et al, OSDI'02)
  - UR-HDFS (Celement et al, SOSP'09)
- Database replication
  - Commit Barrier Scheduling (Vandiver et al, SOSP'07)
  - Byzantium (Garcia et al, EuroSys'11)
- Coordination Service
  - DepSpace (Bessani et al, EuroSys'08)
  - UR-Zookeeper (Clement et al, SOSP'09)
- Naming Services
  - DNS (Cachin & Samar, DSN'04)
  - LDAP (FCUL, unpublished)

BFT Real Applications?

- Tolerating non-malicious Byzantine faults
  - Memory and disk corruptions are relatively common at large scale
  - These problems are detected and corrected using end-to-end integrity checks (i.e., crypto hashes stored separately)
  - Can we use BFT SMR to tolerate this?
    - Where these faults happen?
    - Are there simple techniques?
  - What about software (heisen)bugs?
- General fault tolerance
  - BFT is a general technique for fault tolerance
  - The next step on fault tolerance evolution
- Malicious Byzantine faults
  - What if Byzantine faults are the result of successful attacks?
  - BFT is not enough, we need **Intrusion tolerance**

Intrusion Tolerance (InTol)

- Coined by Joni Fraga and David Powell
  "A Fault- and Intrusion-Tolerant File System", IFIP SEC,1985
- An **intrusion-tolerant system** can maintain its security properties (**confidentiality**, **integrity** and **availability**) despite some of its components being compromised
- **Appeal:** since it's impossible to prove that a system has no vulnerabilities, it is more safe to assume that intrusions can happen

The Promise of BFT

- From PBFT' abstract (Castro & Liskov, OSDI'99):
  "We believe that Byzantine fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior."

InTol vs BFT

- BFT replication protocols are a key mechanism for intrusion-tolerant systems
  - However, I/T systems assume *faults may be caused by malicious and intelligent adversaries*
- Differences and I/T added requirements:
  - Unfavorable executions
  - Diversity
  - Recovery and Self-healing
  - Confidentiality

Intrusion-tolerant Systems

- Definition

  *An intrusion-tolerant system is a replicated system in which a **malicious adversary** needs to compromise more than f out-of n components in less than T time units in order to make it fail.*

  Comments:
  - Similar to BFT with proactive recovery
  - *T* and *f* make little sense without previous requirements
  - Other definitions are possible

Problems of Intrusion Tolerance

- Originally described in (Bessani, WRAITS'11)

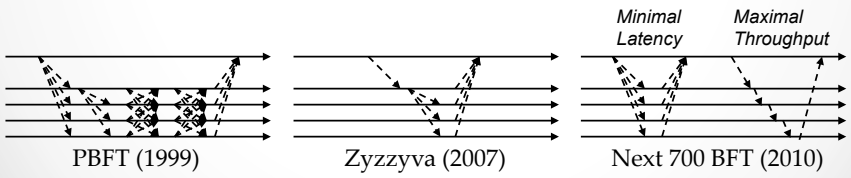
- 3 Solved
- 2 Half-solved
- 5 Open

Solved Problem:
Performance

1990s: first implementations appeared with useful performance (Rampart, SecureRing)

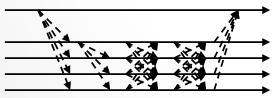
1999: Castro & Liskov' PBFT

2000s: PBFT-like protocols with better performance in certain favorable conditions

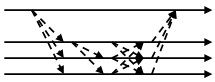


Solved Problem:
Resource Efficiency

- Separating agreement from execution
  - *3f+1* replicas for ordering requests
  - *2f+1* replicas for executing requests
  - *f+1* exec. replicas may be sufficient with VMs
- Trusted components (e.g., TPM)
  - Agreement with 2f+1 replicas (instead of 3f+1)



PBFT



MinBFT

Minimal:
- Number of replicas
- Communication steps
- Trusted component

Solved Problem:
Recovery

- Problem with tolerating *f* faults:
  - If an intelligent adversary is able to compromises *f* machines, given enough time, he/she will compromise *f+1* (or more)
- Solution: Periodic (Proactive) Recovery
  - Replicas (compromised or not) are cleaned periodically
- Requires a trusted real-time component

Half-solved Problem:
Diversity

- *f*-fault-tolerant replicated systems are useful only if faults are not correlated
- It usually requires **diverse replicas**
  - Different administrative domains
  - N-version programming (effective?)
  - Obfuscation, Memory randomization (effective?)
  - Use of different components like databases, file systems, operating systems is effective!
- *What about deploying diversity?*

Half-solved Problem:
Robust Performance of BFT

- BFT replication is
  - very **efficient** in **favorable** conditions
  - very **inefficient** in **unfavorable** conditions
- What about a balance?
  - **efficient enough** in **most** conditions
- Design principles (Prime, Aardvark, AQS)
  - No complex optimizations
  - Use public-key crypto if needed
  - Exploit application semantics for optimizations

Open Problems:
Intrusion Reaction

- Most BFT protocols only tolerate faults and don't take actions against malicious replicas (others than what is required for correctness)
- In practice, replica behavior needs to be monitored and recovery actions need to be executed if intrusions are detected
- **Research question:** *Given the specification of a protocol, how to automatically detect misbehaviors and react to them?*

Open Problems:
Time-bounded State Transfer

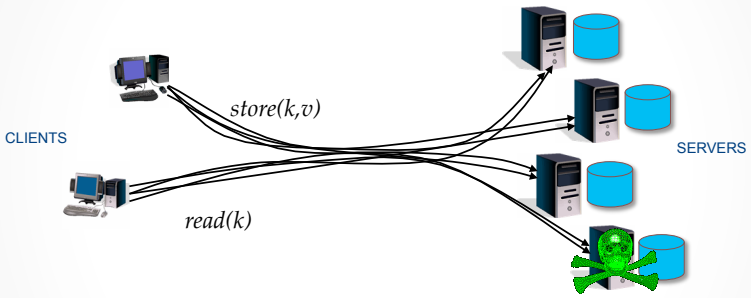
- Recall that the window of vulnerability of an intrusion-tolerant system is bounded by *T*
  - Every *T* time units all replicas are rejuvenated
  - Every replica must take no more than *T/n* time units to recover itself, i.e., take the following steps:
    - Shutdown
    - Chose a clean (and different) OS image
    - Boot
    - **Fetch and validate service state**
- **Research question:** *How to bound the last step?*

Open Problems:
Diversity Management

- **Research question:** *Assume we have a pool of diverse configurations for the system replicas, how to choose the best set?*
  - The idea is to minimize the number of shared vulnerabilities/bugs among any two replicas
  - This is even more complicated if replicas change at runtime
- Besides that, diversity means management of complexity. *How to deal with it?*

 117

Open Problems:
Confidential Operation



- One intrusion → Data leaked
- Threshold crypto/secret sharing help in some cases, e.g., storage systems (Bessani et al, EuroSys'08)
- Homomorphic crypto can be solution

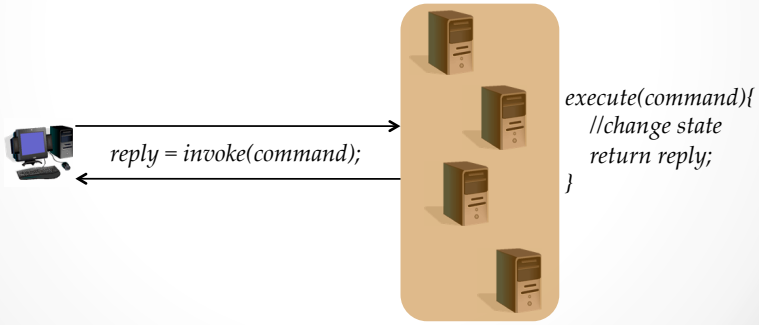
 118

Open Problems:
Graceful Degradation

- Our intrusion tolerance definition is very strict (*all-or-nothing*)
- **Research question:** *How to specify degraded behaviors for intrusion tolerant systems in general?*
- Examples: What if …
  - … there are more than *f* faulty replicas?
  - … the system is completely asynchronous?

SMR Programming Model

- Basic client-server synchronous RPC



SMR Programming Model

- What about server-initiated communication?
  - Client needs to poll the server for updates
- What about asynchronous RPC?
  - Do a synchronous RPC at the client-side on a separated thread
- What about nested calls?
  - Requires special support for the API
- What about multithreading?
  - Remove it!
  - The replication library provides nonces and timestamps for dealing with other sources of non-determinism

BFT-SMaRt

- Started in 2006, as a Byzantine Paxos implementation on the Neko simulator
- Later extended to be the replication layer of DepSpace (Bessani et al, EuroSys'08)
- Currently used/maintained by researchers in Portugal, Brazil and Germany
- Sponsored by:

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Navigators

TClouds

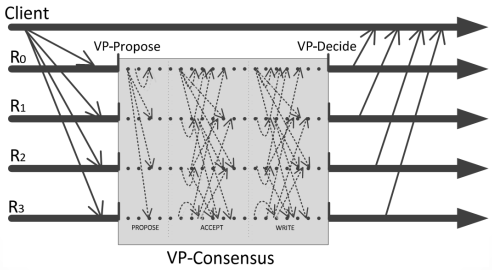
>MASSIF<

SEVENTH FRAMEWORK PROGRAMME

FCT Fundação para a Ciência e a Tecnologia

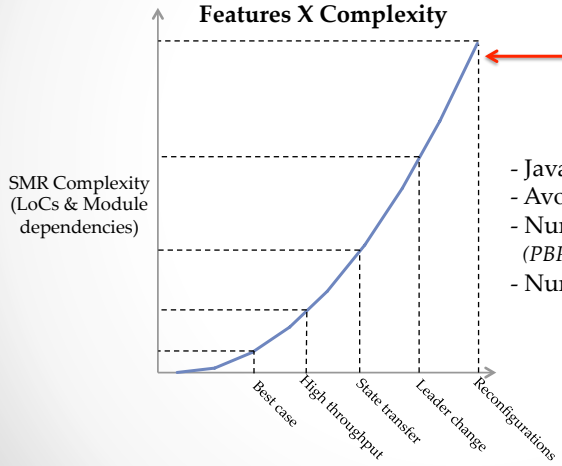
BFT-SMaRt

- BFT-SMaRt design principles:
  - Java-based (for security and correctness reasons)
  - No optimizations that bring complexity
  - Modularity
  - Features: Extensible API, State Management, Reconfiguration



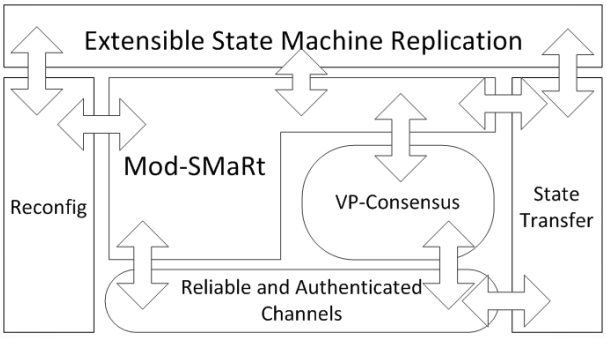
- Implements a protocol very similar to PBFT, but modular; Mod-SMaRt (Sousa & Bessani, EDCC'12)

Dealing with Complexity

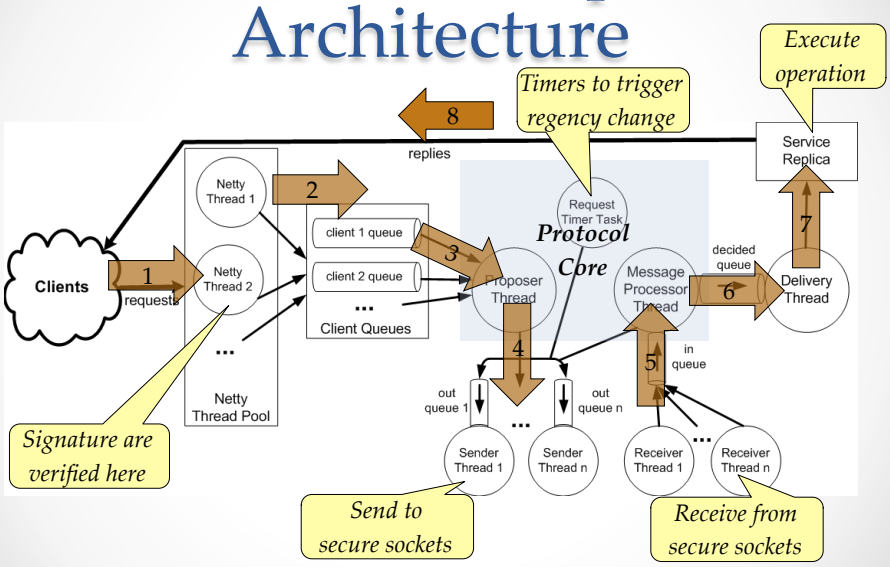


- Java instead of C++
- Avoid overcomplicated optimizations
- Number of lines of code: 8399
  (PBFT: ~20K LoCC; UpRight: ~22K LoJC)
- Number of classes/interfaces: 90

Modularity



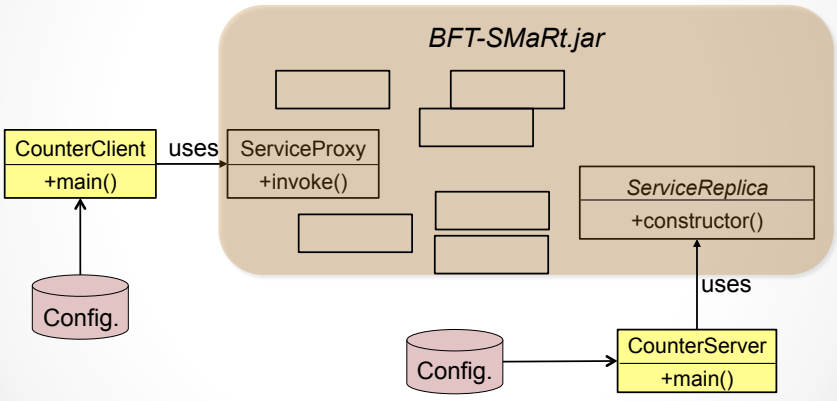
BFT-SMaRt Replica Architecture



BFT-SMaRt Software

- It is a library (*.jar* file) that must be linked with the client and the servers…
- There is no service/component that must be deployed or managed besides the BFT client and server
- Available at http://code.google.com/p/bft-smart/
- Current version: 0.7
  - Many disruptive features are being integrated in the code
  - API changes will happen
  - Bugs remain
  - Any help is welcome!

BFT-SMaRt Software



Configuration

- A directory containing three things
  - The *keys* directory with the process *i privatekeyi* file and *publickeyj* for every other process *j*
    - In the future these keys will go to keystores/trustores
  - *hosts.config*: IP:port of the n replicas

```
#id address port (0 to n-1 are replicas)
0 127.0.0.1 11000
1 127.0.0.1 11010
2 127.0.0.1 11020
3 127.0.0.1 11030
```

  - Do not use consecutive ports (each replica uses its port p, plus p+1)

Configuration

- *system.config*: a Java properties file containing the system parameters

```
system.authentication.hmacAlgorithm = HmacSHA1
system.servers.num = 4
system.servers.f = 1
system.totalordermulticast.timeout = 12000000
system.totalordermulticast.highMark = 10000
system.totalordermulticast.maxbatchsize = 400
system.totalordermulticast.verifyTimestamps = false
system.totalordermulticast.state_transfer = true
system.totalordermulticast.checkpoint_period = 50
system.totalordermulticast.revival_highMark = 10
system.communication.useSignatures = 0
system.communication.useMACs = 1
system.initial.view = 0,1,2,3
system.debug = 0
```

BFT-SMaRt Programming

- Client-side:
  - **ServiceProxy** is the main class to be used
  - Requests and replies are byte arrays (to avoid unnecessary overheads)

```
public class ServiceProxy extends ... {
          public ServiceProxy(int processId) ...
   public ServiceProxy(int processId,
          String configHome,
          Comparator<byte[]> replyComparator,
          Extractor replyExtractor) ...

   public byte[] invokeOrdered(byte[] request) ...
   public byte[] invokeUnordered(byte[] request) ...
   public void invokeAsynchronous(byte[] request,
          ReplyListener listener, int[] targets) ...
}
```


BFT-SMaRt Programming

- Server-side:
  - **ServiceReplica** is the main class
  - It needs an implementation of Executable and Recoverable to work

```
public class ServiceReplica extends ... {
   public ServiceReplica(int id,
          Executable executor,
          Recoverable recover) ...
   public ServiceReplica(int id, String configHome,
          boolean isToJoin, Executable executor,
          Recoverable recover) ...

   public void leave() ...
   public ReplicaContext getReplicaContext () ...
}
```

BFT-SMaRt Programming

- Server-side (cont.):

```
public interface Executable {
    public byte[] executeUnordered(byte[] command,
            MessageContext msgCtx);
}
public interface SingleExecutable extends Executable {
    public byte[] executeOrdered(byte[] command,
            MessageContext msgCtx);
}
public interface BatchExecutable extends Executable {
    public byte[][] executeBatch(byte[][] command,
            MessageContext[] msgCtx);
}
public interface Recoverable {
    public byte[] getState();
    public void setState(byte[] state);
}
```


Creating an In-Memory KV-Store with BFT-SMaRt

- Download BFT-SMaRt 0.7 from http://code.google.com/p/bft-smart
- (optional) Create a project in your favorite Java IDE and add *dist/BFT-SMaRt.jar* and other *lib/*.jar* to it
- Create a *KVMessage* class to represent the messages exchanged between clients and the replicas
- Create a *KVServer* class implementing the *SingleExecutable* and *Recoverable* interfaces, and using *ServiceReplica*
- Create a *KVClient* class using *ServiceProxy*

References

- Kubiatowicz et al. OceanStore: An Architecture for Global-scale Persistent Storage. ASPLOS'00
- Adya et al. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. OSDI'02
- Vandiver et al. Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. SOSP'07
- Garcia et al. *Efficient Middleware for Byzantine Fault-tolerant Database Replication.* EuroSys'11
- Bessani et al. DepSpace: A Byzantine Fault-tolerant Coordination Services. EuroSys'08
- Christian Cachin and Asad Samar. Secure distributed DNS. DSN'04
- Fraga & Powel. A Fault- and Intrusion-Tolerant File System, IFIP SEC'85
- Bessani. From Byzantine Fault Tolerance to Intrusion Tolerance (A position paper). WRAITS'11
- Sousa & Bessani. From Byzantine Consensus to BFT State Machine Replication: A latency-optimal transformation. EDCC'12

http://www.di.fc.ul.pt/~bessani

http://code.google.com/p/bft-smart

Navigators

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

EuroSys 2012

