Paper Presentation ECS 265

Fault Scalable Byzantine Fault Tolerant Services

By Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, Jay J. Wylie

Presented by Zainub Sheikh, Apratim Shukla, Kshitij Sinha, Adesh Thakare, Warren Wnuck

October 10th Fall 2022

Department of Computer Science

UC Davis

Recap

- What is Byzantine General Tolerance Problem?
  - Consensus problem with malicious generals or messengers
- What is BFT?
  - An algorithm to allow nodes to come to a consensus with byzantine nodes in the system
    - Fail-Stop Fault
    - Byzantine Fault
- What is PBFT?
  - Agreement-based
  - Synchronous
- Problems with distributed database systems

Query/Update Protocol

- What is Query/Update Protocol (Q/U)?
  - Tool aims to solve network faults and fault tolerance by approaching a fault-scalable optimistic asynchronous quorum-based approach to building a byzantine fault tolerant service
- Quorum
- Asynchronous
- Fault-Scalable
- Optimistic

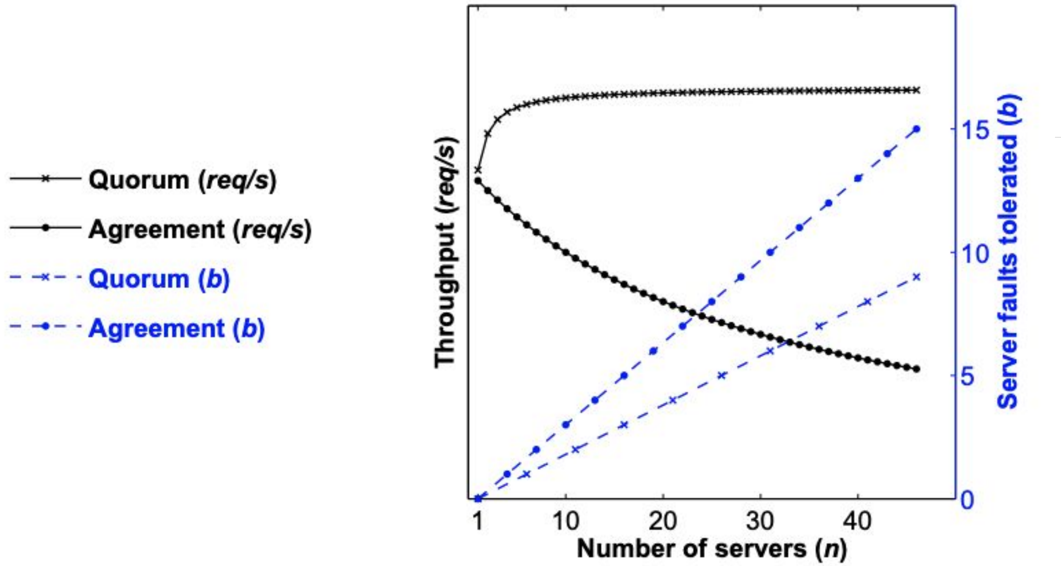
Attributes of Q/U Protocol

- Needs 5f+1 nodes where f is number of byzantine faults being tolerated
- Query in Q/U
  - A deterministic method which doesn’t modify objects
- Update in Q/U
  - Modifies objects
- Others related to the techniques
  - Able to update multiple objects, timestamps objects, has an operations-based interface, less server-to-server communication, less phases (1-2 phases than 3-4)
- Results in Q/U counter object achieved 36% performance decrease with 1-5 faults tolerated compared to a BFT-based counter object which achieved 83% performance decrease

Throughput-Scalability and Efficiency

BFT Agreement-Based	**BFT Quorum-Based**
Every Server Processes each request	Only quorums of servers process each request
Performs server-to-server broadcast	Servers-to-server communication is avoided
Increasing number of servers does not increase throughput	Throughput degrades gradually as more servers are tolerated
More servers means less useful work per server	More servers, larger quorum size, more costly
Requires fewer servers	Requires more servers

Throughput Quorum-Based vs Agreement-Based



Efficiency

- Efficiency from optimism
    - Failure and Concurrency free, queries and updates happen in one phase
    - Failure atomicity
        - Pessimistic protocols utilize 2 phases at least
        - Optimistic does not have prepare phase and Q/U does not have a sense of commit phase
    - Concurrency atomicity
        - Pessimistic protocols rely on the primary or having locks
        - Optimistic protocols rely on versioning servers
            - Livelock can occur
- System efficiency
    - Caching versions, updates, preferred quorums, not needing to agree

Throughput-Scalability

- Primary benefit of Q/U is the quorum-based approach to fault-scalability
- Also with throughput-scalability
    - More servers than what is needed while providing fault-tolerance, will increase throughput
- Past reviewers of quorum’s approach to throughput-scalability is hard to achieve but fail to look at different attributes that Q/U solves
    - Jiménez-Peris et al. claim other approach is better for database applications
        - Q/U provides service replication, asynchronous, and failure and concurrency atomicity
    - Gray et al. claim data replication with quorum-based leads to scaleup pitfall
        - Fine-grained Q/U objects
        - Example with Q/U-NFSv3 metadata service with fine-grained Q/U objects

The Query/Update Protocol

System Model

- An asynchronous timing model is used.

  State of the server:

  - A server is **benign** if it is correct or if it follows its specification except for crashing and (potentially) recovering. Basically, NOT MALICIOUS. Otherwise the server is called **malevolent.** Malevolent is actually used to specify those type of servers that exhibit out of specification, non-crash behaviour.
  - A server is called faulty if it crashes and does not recover or is malevolent.

Some assumptions...

We assume a universe of servers U.

- |U| = n
- $\mathcal{T} \subseteq 2^U$ and $\mathcal{B} \subseteq 2^U$
  This is the power set of U.
- T$\epsilon\mathcal{T}$ are faulty servers
- B$\epsilon\mathcal{B}$ are malevolent servers

Note*
In a threshold quorum system, a fail prone system can be defined by bounds to the number of faulty servers. There are no more than t faulty servers out of which $b \le t$ are malevolent.

From the definition of faulty and malevolent servers, it follows that $B \subseteq T$.

In this quorum-based protocol, a quorum is set of subsets of U, every pair of which intersect. In short it is the subset of the power set of the servers.

Terminology

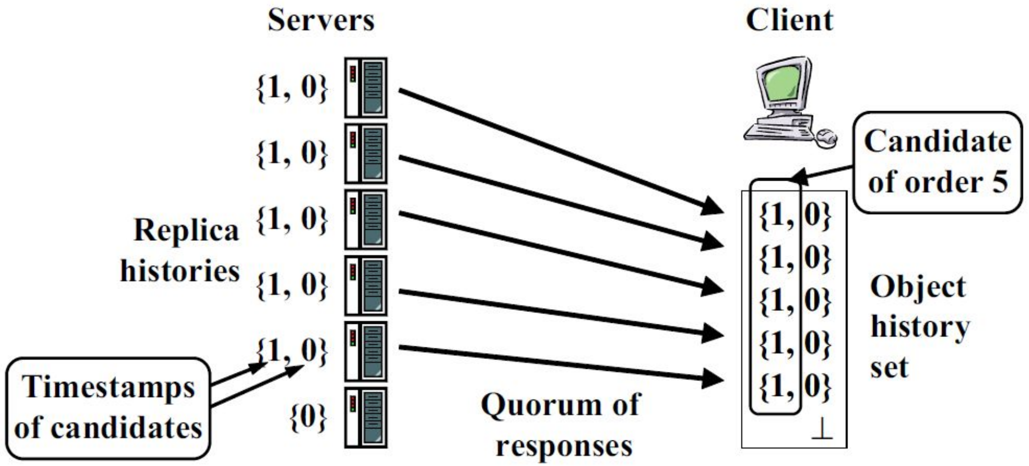
- **Queries** - These are the read only methods that consists of arguments.
- **Updates** - These are the methods that modify an object’s state.
- **Quorum** - subset of servers. In an asynchronous system, clients only receive responses from a subset of servers.
- **OHS** - object history set - the array of replica histories indexed by the server.
- **Candidates** - Timestamps in the relica history are referred to as candidates. Each candidate has a corresponding object version.
- **Conditioned-on OHS** - The client’s operation *condition on* the OHS and
- **Candidate order -** the number of replica histories in the object history set in which it appears.
- **Replica History -** The version generated by the server after invoking the method.

Workflow:

Client side	Client and Server	Server side
Client issues a request to the quorum for performing an operation. The client places the method to invoke, and the arguments for the method and sends it to the server. This communicates the information about global system state.	Client and server both classify to verify the versions on which the operation needs to be performed.	The server receives requests and if it accepts the request, it invokes the method on its replica. This creates new object version. The server retains the object version and its associated timestamp. It returns this OHS to the client.

UCDAVIS

Example of the client and server states



Optimistic Nature of the Q/U protocol

- It allows the client to complete queries, and updates conditioned on a cached OHS in a single phase of client server communication.
- This allows the client to avoid receiving the OHS before every update operation.
- This is during failure and concurrency-free access.

UCDAVIS

Preferred quorum

To promote locality of access which increases efficiency, each object has a *preferred quorum* at which clients try to access first. In case of server failures, clients may have to access a *non-preferred quorum*.

If a client accesses a non-preferred quorum, some servers that receive requests may not have a conditioned-on object version. In this case, an *object sync* needs to be performed. This helps in retrieving conditioned-on object version.

Responses from at least b+1 host servers are required to validate that the data it receives is correct.

Authenticators

Each server pairs and authenticator and clients include authenticators in their conditioned-on OHS. Authenticators are a list of HMACs that prevent malevolent clients from fabricating replica histories for benign servers.

HMACs - Hash based message authentication code

Necessary because servers do not exchange replica histories with one another, but through the client.

Validation

- Validation is done by comparing the timestamps of the candidates.
- Currentness needs to be validated so that the operation is performed on the latest OHS.
- The server accepts the request only if the validation passes.

Concurrency and Repair

Only update of a specific object version can be completed. As such concurrent accesses may fail and result in replica histories at different servers that have a different latest candidate.

Repair is performed when an object is accessed concurrently and it may FAIL which may result in replica histories at different servers have different latest candidates

Repair has two main steps:

- Barrier - Allows to safely advance in logical time.
- Copy - Copies the latest object version prior to the barrier formed in logical time.

Repair, i.e. performing barrier and copy, is a fundamental aspect of Q/U protocol.

Classification and constraints

Classification of an OHS dictates whether the latest candidate is complete, repairable or incomplete. It helps in deciding whether repair is required or not.

Constraints on the quorum system, in conjunction with classification rules, ensure that established candidates are classified as repairable or complete. This ensures that the update are invoked on the latest object version

Given a set of server responses S that share the same candidate, the classification rules are:

$$\textbf{classify}(S) = \begin{cases} \textit{complete} & \text{if } \exists Q \in \mathcal{Q} : Q \subseteq S, \\ \textit{repairable} & \text{if } (\forall Q \in \mathcal{Q} : Q \not\subseteq S) \wedge \\ & (\exists Q \in \mathcal{Q}, R \in \mathcal{R}(Q) : \\ & R \subseteq S), \\ \textit{incomplete} & \text{otherwise.} \end{cases} \quad (1)$$

The constraints that we require are as follows:

$$\forall Q \in \mathcal{Q}, \forall T \in \mathcal{T} : Q \cup T \subseteq U; \quad (2)$$

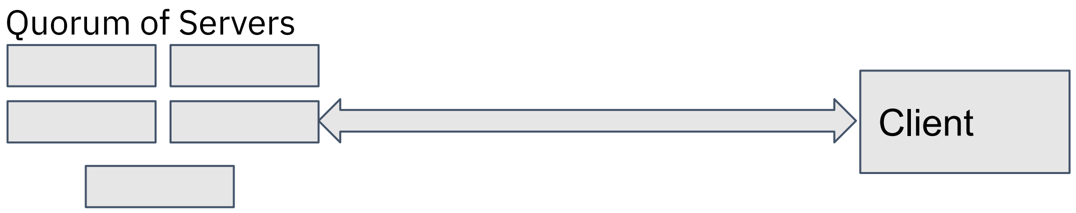
$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \exists R \in \mathcal{R}(Q_i) : R \subseteq Q_i \cap Q_j \setminus B; \quad (3)$$

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \forall R \in \mathcal{R}(Q_j) : Q_i \cap R \not\subseteq B. \quad (4)$$

Implementation Details & System Design

Cached object history set

1. Clients caches the OHS for the objects they access regularly.
2. If OHS that client has in cache is not updated compared to server, and if client sends request for object, then first server gives the client the latest OHS from replica and it gets updated on client side and client OHS becomes more current, then the server sends the requested object.



Optimistic query execution

1. If a client has not accessed an object recently or is accessing for the first time, it's cached OHS may not be current.
2. It is still possible for a query to complete in a single phase (only during failure and concurrency - free access)
3. Servers, noting that the conditioned-on OHS is not current, invoke the query method on the latest object version they store. The server, in its response, indicates that the OHS is not current, but also includes the answer for the query invoked on the latest object version and its replica history. After the client has received a quorum of server responses, it performs classification on its object history set. Classification allows the client to determine if the query was invoked on the latest complete object version; if so the client returns the answer.

Quorum access strategy and probing

1. Clients access an object’s preferred quorum.
2. Clients initially send requests to the preferred quorum. If responses from the preferred quorum are not received in a timely fashion, the client sends requests to additional servers.
3. Accessing a nonpreferred quorum requires that the client probe to collect a quorum of server responses. Servers contacted that are not in the preferred quorum will likely need to object sync.
4. A deterministic function is used to map an object’s ID to its preferred quorum. A simple policy of assigning preferred quorums based on the object ID modulo n is used in this paper.

Inline repair

1. Repairs a candidate “in place” at its timestamp.

2. Inline repair is useful in the face of server failures that lead to non-preferred quorum accesses. The first time an object is accessed after a server in its preferred quorum has failed, the client can likely perform inline repair at a non-preferred quorum.

3. The first time an object is accessed after a server in its preferred quorum has failed, the client can likely perform inline repair at a non-preferred quorum.

Handling repeated requests at the server

1. Server may receive multiple requests at the same time from multiple clients or from same client multiple times.
2. No matter how many times request is received, it has to be invoked the method on object once and provide the same results every time.
3. Before checking if the conditioned-on OHS is current, the server checks to determine if a candidate is already in its replica history with the timestamp it setup. If yes, the server retrieves the corresponding answer and returns immediately.

Retry and backoff policies

1. Update-update concurrency among clients leads to contention for an object. (competition for an object is resolved by a random exponential backoff policy)
2. To perform an object sync, a server uses the conditioned-on object history set to identify servers from which to solicit the object state. Like accessing a preferred quorum, if some responses are not forthcoming (or do not all match) additional servers are probed.

https://cloud.google.com/iot/docs/how-tos/exponential-backoff#:~:text=An%20exponential%20backoff%20algorithm%20retries,seconds%20and%20retry%20the%20request.

Object Sync

1. When server finds that the OHS it has is outdated compared to what it got from client request, it searches for that updated object on the other servers.

2. Like accessing a preferred quorum, if responses are received (or do not all match) additional servers are probed.

3. They also copy the OHS other servers have currently, only a single correct server need send the entire object version state.

Authenticators & Timestamps

1. Authenticators consist of HMACs.
2. One for each server in the system.
3. The use of n HMACs in the authenticator, in place of a digital signature, is based on the scale of systems, even though the cost of digital signs does not increase as number of servers grows.
4. Timestamps include the operation and the object history set, and is sent with the request along with authentication protocol.

Compact replica histories

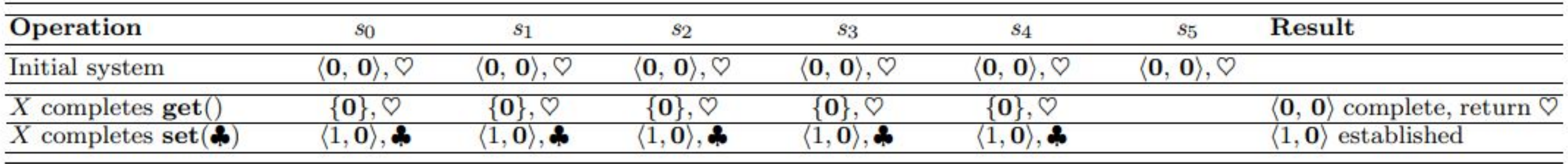
1. Servers need only return the portion of their replica histories with timestamps greater than the conditioned-on timestamp of the most recent update the client had.
2. Server searches it replica history and returns only those replicas for which the timestamp is greater than the current timestamp sent by client.
3. Object versions corresponding to candidates pruned from a server’s replica history could be deleted. Doing so can free storage space on servers. However, deleting past object versions could make it impossible for a server to respond to slow repeated requests from clients and slow object sync requests from other servers. As such, there is a practical trade-off between reclaiming storage space on servers and ensuring that clients do not need to abort operations.

Malevolent components

1. Malevolent components can affect the ability of correct clients to make progress. However, the Q/U protocol can be modified to ensure that isolated correct clients can make progress.
2. Malevolent clients may not follow the specified backoff policy. For contended objects, this is a form of denial-of-service attack. However, additional server-side code could limit clients with a request rate limit to prevent DDOS.
3. Malevolent clients have some impact on performance. Malevolent clients may issue updates only to subsets of a quorum. This results in the system, requiring correct clients to perform repair. Techniques such as lazy verification, in which the correctness of client operations is verified in the background and in which limits are placed on the amount of unverified work a client may inject in the system, can control the impact that such malevolent clients have on performance.
4. They can also mess up with HMACS, If a malevolent server corrupts more than b HMACs in its authenticator, then a client can conclude, from the responses of other servers, that the server is malevolent and exclude it from the set of servers it contacts, this is sufficient for the client to solicit more responses from additional servers and make progress.

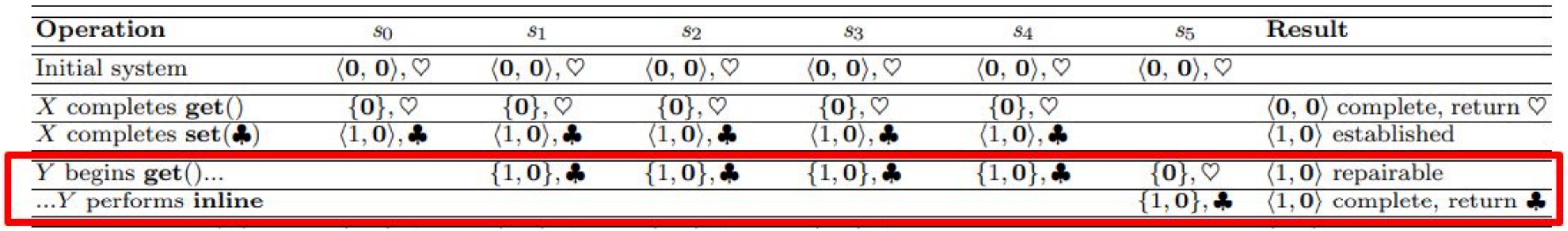
Example Q/U protocol execution

- The example is for an object that exports two methods: get (a query) and set (an update).
- The object can take on four distinct values (♣, ♦, ❤, ♠) and is initialized to ❤
- Operations are performed by the three clients: X, Y , and Z
- Clients X and Z interact with the object's preferred quorum (the first five servers) and Y with a non-preferred quorum (the last five servers).
- s0, ..., s5 are the six benign servers in the example.



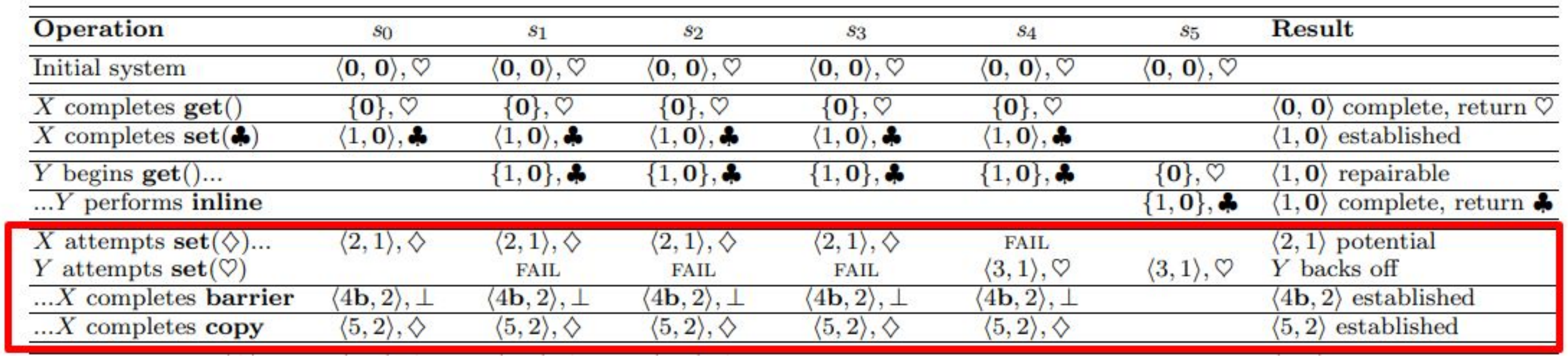
Operation	s_0	s_1	s_2	s_3	s_4	s_5	Result
Initial system	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	
X completes **get**()	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$		$\langle \mathbf{0}, \mathbf{0} \rangle$ complete, return $\heartsuit$
X completes **set**($\clubsuit$)	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$		$\langle 1, \mathbf{0} \rangle$ established

- In the above example, the middle columns list candidates stored by and replies (replica histories or status codes) returned by six benign servers (s0, ..., s5).
- The right column lists if updates yield an established or potential candidate (assuming all servers are benign) and the results of classification for queries.
- The first sequence demonstrates failure and concurrency free execution: client X performs a get and set that each complete in a single phase.



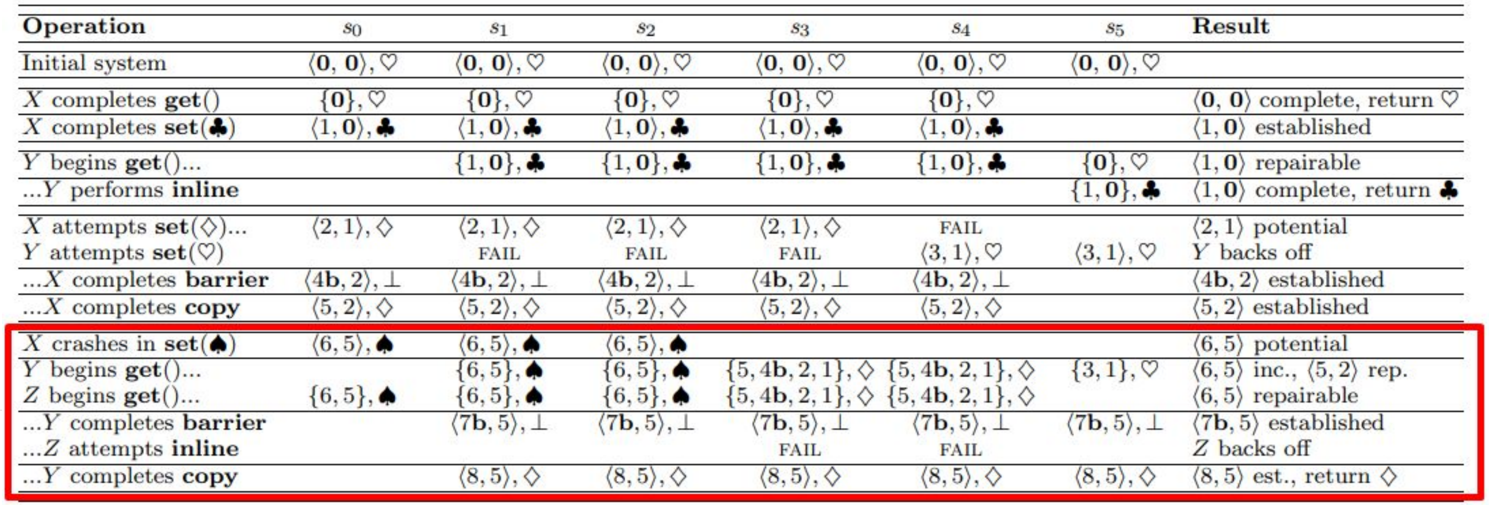
Operation	s_0	s_1	s_2	s_3	s_4	s_5	Result
Initial system	⟨**0**, **0**⟩, ♡	⟨**0**, **0**⟩, ♡	⟨**0**, **0**⟩, ♡	⟨**0**, **0**⟩, ♡	⟨**0**, **0**⟩, ♡	⟨**0**, **0**⟩, ♡	
X completes **get**()	{**0**}, ♡	{**0**}, ♡	{**0**}, ♡	{**0**}, ♡	{**0**}, ♡		⟨**0**, **0**⟩ complete, return ♡
X completes **set**(♣)	⟨1, **0**⟩, ♣	⟨1, **0**⟩, ♣	⟨1, **0**⟩, ♣	⟨1, **0**⟩, ♣	⟨1, **0**⟩, ♣		⟨1, **0**⟩ established
Y begins **get**()...		{1, **0**}, ♣	{1, **0**}, ♣	{1, **0**}, ♣	{1, **0**}, ♣	{**0**}, ♡	⟨1, **0**⟩ repairable
...Y performs **inline**						{1, **0**}, ♣	⟨1, **0**⟩ complete, return ♣

- Client Y performs a get in the second sequence that requires repair. Since there is no contention, Y performs inline repair. Server s5 performs object syncing to process the inline repair request.



Operation	s_0	s_1	s_2	s_3	s_4	s_5	**Result**
Initial system	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	
X completes **get**()	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$		$\langle \mathbf{0}, \mathbf{0} \rangle$ complete, return $\heartsuit$
X completes **set**($\clubsuit$)	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$		$\langle 1, \mathbf{0} \rangle$ established
Y begins **get**()...		$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{\mathbf{0}\}, \heartsuit$	$\langle 1, \mathbf{0} \rangle$ repairable
...Y performs **inline**						$\{1, \mathbf{0}\}, \clubsuit$	$\langle 1, \mathbf{0} \rangle$ complete, return $\clubsuit$
X attempts **set**($\diamondsuit$)...	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	FAIL		$\langle 2, 1 \rangle$ potential
Y attempts **set**($\heartsuit$)		FAIL	FAIL	FAIL	$\langle 3, 1 \rangle, \heartsuit$	$\langle 3, 1 \rangle, \heartsuit$	Y backs off
...X completes **barrier**	$\langle \mathbf{4b}, 2 \rangle, \perp$	$\langle \mathbf{4b}, 2 \rangle, \perp$	$\langle \mathbf{4b}, 2 \rangle, \perp$	$\langle \mathbf{4b}, 2 \rangle, \perp$	$\langle \mathbf{4b}, 2 \rangle, \perp$		$\langle \mathbf{4b}, 2 \rangle$ established
...X completes **copy**	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$		$\langle 5, 2 \rangle$ established

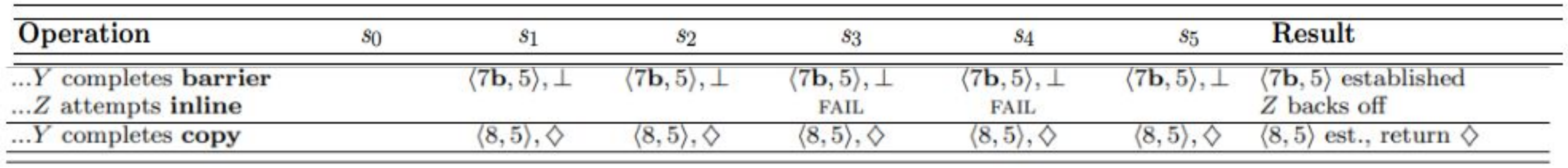
- In the third sequence, concurrent updates by X and Y are attempted. Contention prevents either update from completing successfully.
- At server s4, the set from Y arrives before the set from X. As such, it returns its replica history <3, 1>, <1, 0> with FAIL.
- The candidate <3, 1> in this replica history dictates that the timestamp of the barrier be 4b. For illustrative purposes, Y backs off. Client X subsequently completes barrier and copy operations.



Operation	s_0	s_1	s_2	s_3	s_4	s_5	Result
Initial system	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	$\langle\mathbf{0},\mathbf{0}\rangle,\heartsuit$	
X completes **get**()	$\{\mathbf{0}\},\heartsuit$	$\{\mathbf{0}\},\heartsuit$	$\{\mathbf{0}\},\heartsuit$	$\{\mathbf{0}\},\heartsuit$	$\{\mathbf{0}\},\heartsuit$		$\langle\mathbf{0},\mathbf{0}\rangle$ complete, return $\heartsuit$
X completes **set**($\clubsuit$)	$\langle 1,\mathbf{0}\rangle,\clubsuit$	$\langle 1,\mathbf{0}\rangle,\clubsuit$	$\langle 1,\mathbf{0}\rangle,\clubsuit$	$\langle 1,\mathbf{0}\rangle,\clubsuit$	$\langle 1,\mathbf{0}\rangle,\clubsuit$		$\langle 1,\mathbf{0}\rangle$ established
Y begins **get**()...		$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{\mathbf{0}\},\heartsuit$	$\langle 1,\mathbf{0}\rangle$ repairable
...Y performs **inline**						$\{1,\mathbf{0}\},\clubsuit$	$\langle 1,\mathbf{0}\rangle$ complete, return $\clubsuit$
X attempts **set**($\diamondsuit$)...	$\langle 2,1\rangle,\diamondsuit$	$\langle 2,1\rangle,\diamondsuit$	$\langle 2,1\rangle,\diamondsuit$	$\langle 2,1\rangle,\diamondsuit$	FAIL		$\langle 2,1\rangle$ potential
Y attempts **set**($\heartsuit$)		FAIL	FAIL	FAIL	$\langle 3,1\rangle,\heartsuit$	$\langle 3,1\rangle,\heartsuit$	Y backs off
...X completes **barrier**	$\langle\mathbf{4b},2\rangle,\bot$	$\langle\mathbf{4b},2\rangle,\bot$	$\langle\mathbf{4b},2\rangle,\bot$	$\langle\mathbf{4b},2\rangle,\bot$	$\langle\mathbf{4b},2\rangle,\bot$		$\langle\mathbf{4b},2\rangle$ established
...X completes **copy**	$\langle 5,2\rangle,\diamondsuit$	$\langle 5,2\rangle,\diamondsuit$	$\langle 5,2\rangle,\diamondsuit$	$\langle 5,2\rangle,\diamondsuit$	$\langle 5,2\rangle,\diamondsuit$		$\langle 5,2\rangle$ established
X crashes in **set**($\spadesuit$)	$\langle 6,5\rangle,\spadesuit$	$\langle 6,5\rangle,\spadesuit$	$\langle 6,5\rangle,\spadesuit$				$\langle 6,5\rangle$ potential
Y begins **get**()...		$\{6,5\},\spadesuit$	$\{6,5\},\spadesuit$	$\{5,\mathbf{4b},2,1\},\diamondsuit$	$\{5,\mathbf{4b},2,1\},\diamondsuit$	$\{3,1\},\heartsuit$	$\langle 6,5\rangle$ inc., $\langle 5,2\rangle$ rep.
Z begins **get**()...	$\{6,5\},\spadesuit$	$\{6,5\},\spadesuit$	$\{6,5\},\spadesuit$	$\{5,\mathbf{4b},2,1\},\diamondsuit$	$\{5,\mathbf{4b},2,1\},\diamondsuit$		$\langle 6,5\rangle$ repairable
...Y completes **barrier**		$\langle\mathbf{7b},5\rangle,\bot$	$\langle\mathbf{7b},5\rangle,\bot$	$\langle\mathbf{7b},5\rangle,\bot$	$\langle\mathbf{7b},5\rangle,\bot$	$\langle\mathbf{7b},5\rangle,\bot$	$\langle\mathbf{7b},5\rangle$ established
...Z attempts **inline**				FAIL	FAIL		Z backs off
...Y completes **copy**		$\langle 8,5\rangle,\diamondsuit$	$\langle 8,5\rangle,\diamondsuit$	$\langle 8,5\rangle,\diamondsuit$	$\langle 8,5\rangle,\diamondsuit$	$\langle 8,5\rangle,\diamondsuit$	$\langle 8,5\rangle$ est., return $\diamondsuit$

- In the fourth sequence, X crashes during a set operation and yields a potential candidate. The remaining clients Y and Z perform concurrent get operations at different quorums; this illustrates how a potential candidate can be classified as either repairable or incomplete.

UCDAVIS



Operation	s_0	s_1	s_2	s_3	s_4	s_5	Result
...Y completes **barrier**		$\langle \mathbf{7b}, 5\rangle, \perp$	$\langle \mathbf{7b}, 5\rangle, \perp$	$\langle \mathbf{7b}, 5\rangle, \perp$	$\langle \mathbf{7b}, 5\rangle, \perp$	$\langle \mathbf{7b}, 5\rangle, \perp$	$\langle \mathbf{7b}, 5\rangle$ established
...Z attempts **inline**				FAIL	FAIL		Z backs off
...Y completes **copy**		$\langle 8, 5\rangle, \diamond$	$\langle 8, 5\rangle, \diamond$	$\langle 8, 5\rangle, \diamond$	$\langle 8, 5\rangle, \diamond$	$\langle 8, 5\rangle, \diamond$	$\langle 8, 5\rangle$ est., return $\diamond$

- Y and Z concurrently attempt a barrier and inline repair respectively. In this example, Y establishes a barrier before Z’s inline repair requests arrive at servers s3 and s4. Client Z aborts its inline repair operation.
- Subsequently, Y completes a copy operation and establishes a candidate <8, 5> that copies forward the established candidate <5, 2>.

Multi-object updates

- The Q/U protocol allows some updates to span multiple objects. A multi-object update atomically updates a set of objects.
- To perform such an update, a client includes a conditioned on OHS for each object being updated. The set of objects and each object’s corresponding conditioned-on OHS, are referred to as the multi-object history set (multi-OHS).

Correctness

- In the Q/U protocol, operations completed by correct clients are strictly serializable.
- Given the crash-recovery server fault model for benign servers, progress cannot be made unless sufficient servers are up. This means that services implemented with the Q/U protocol are not available during network partitions, but become available and are correct once the network merges.
- The liveness property of the Q/U protocol is fairly weak: it is possible for clients to make progress (complete operations). Under contention, operations (queries or updates) may abort so it is possible for clients to experience livelock.

Evaluation

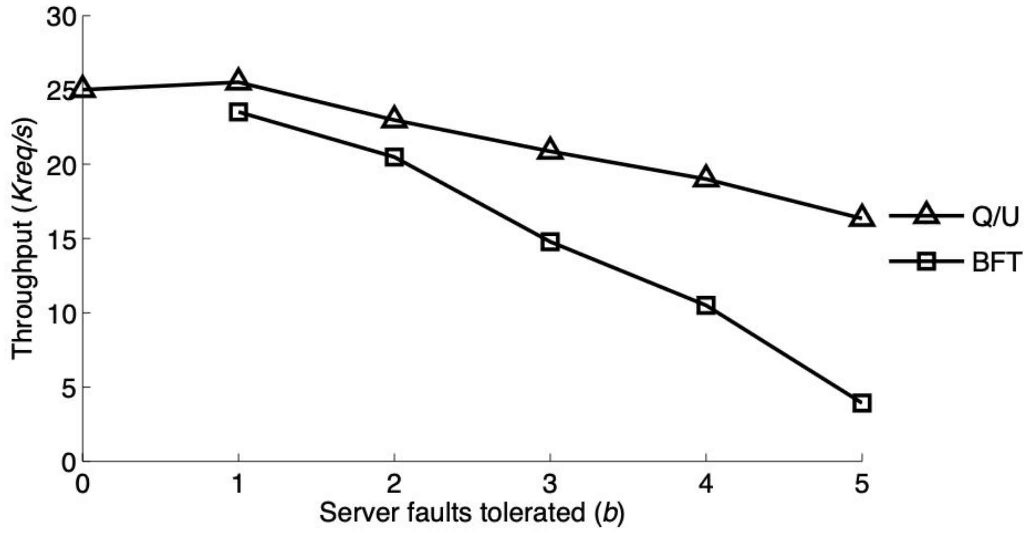


Figure 4: Fault-scalability.

Evaluation Con't

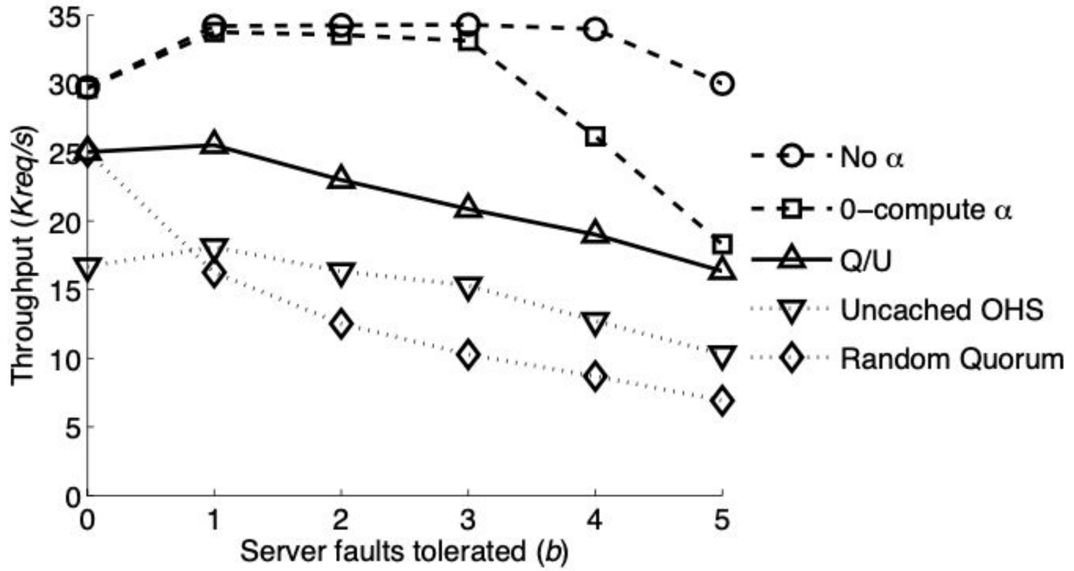


Figure 5: Details of Q/U protocol fault-scalability.

Related Work

- Efficient cryptography
- Byzantine faulty clients
- Agreement based protocols
- Quorum based protocols

Conclusion

- Distributed Systems
- Blockchain
- Consensus
- BFT
- Query Update Protocol
    - Tolerate Byzantine faults
    - Quorum based as opposed to Agreement based
    - Better performance

UCDAVIS

Thank you everyone !

Q/A

ECS 265 Fall 2022 UCD

UCDAVIS
UNIVERSITY OF CALIFORNIA