Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage

Fabíola Greve

Computer Science Department
Federal University of Bahia (UFBA), Brazil

Jointly with:

Eduardo Alchieri, Brasília University (UnB), Brazil

Alysson Neves Bessani, Faculdade de Ciências, Universidade de Lisboa, Portugal

Joni Fraga, Federal University of Santa Catarina (UFSC), Brazil

Agenda

- Motivation
- System Model, Assumptions, System Properties
- View Generators
- Protocol for System Reconfiguration
- Experiments
- Conclusion

Introduction

- Implement fault-tolerant shared memory in dynamic distributed systems
  - Shared memory : Quorum systems
  - Dynamic system: Reconfiguration
- Quorum-based protocols for R/W operations are appealing
  - R/W operations are executed in a quorum of servers
- Ensure consistency and availability of data stored in replicated servers
  - Consistency → Quorum intersections: each quorum of servers intersect
  - Availability → There is always a quorum composed by correct servers

Reconfiguration

- Reconfiguration is the process of changing the set of servers that comprise the system
  - Allows the administrator to deploy or to remove machines at runtime
    - Addition: to deal with increasing workloads
    - Remove: replace old machines
  - Improves system resilience
    - By removing faulty servers
  - Allows its use in many systems where, by their very nature, the set of process that compose the system may change during its execution
    - E.g.: MANETs, P2P

Previous Consensus-Based Solutions

- Dynamic quorum systems, relying on consensus for reconfigurations
- Processes agree on the set of servers (view) supporting the storage
- Rambo [Dist. Comput. 2010]
  - Crash failure model
- Framework proposed by Martin and Alvisi [DSN 2004]
  - Byzantine failure model
- Consensus is not necessary!
- Atomic shared memory emulation in static systems is possible without consensus, ABD [JACM, 1995]

Previous Consensus-Free Solutions

- DynaStore [JACM, 2011]
  - \+ Crash-tolerant dynamic memory that does not rely on consensus for reconfigurations
  - – But, Reconfigurations and R/W protocols are strongly tied
  - – Performance worst than consensus-based solutions in synchronous executions
- SmartMerge [DISC, 2015] and SpSn [DISC 2015]
  - \+ Separate reconfiguration and R/W protocols
  - – But, they do not fully decouple them: for each R/W operation it is necessary to check for view updates
  - – The design decision has a huge impact on performance

Our Contribution: **FreeStore**

- Set of algorithms for implementing dynamic fault-tolerant atomic storage
- FreeStore decouples the execution of R/W protocols from reconfigurations, significatively improving system performance

1. R/W Protocols
   - Can be adapted from R/W protocols, even static protocols, e.g., ABD [JACM, 1995]
2. Reconfiguration Protocols (view updates)
   - *View Generators*: capture agreement requirements for generating new views
   - Protocols to install generated views

Our Contribution: **FreeStore**

- Modularity
  - Separation of concerns
  - The *View Generators* abstraction to capture agreement requirements
- Efficiency
  - Less communication steps than consensus-based and consensus-free counterparts
- Simplicity
  - Novel reconfiguration strategy that reduces the number of intermediary installed views that a process must trasverse before reaching a good view of updates

System Model and Assumptions

- Asynchronous distributed system composed by a universe U of processes, that can be divided in two subsets:
  - An infinite set Π of Servers (each view contains $2f+1$ servers to tolerate up to f failures, and uses quorums of $f+1$ serves)
  - An infinite set C of Clients
  - Clients and Servers are prone to crash failures; Channels are reliable
- Infinite arrivals model with unknown but bounded concurrency
- Views: *membership* composed by a set of tuples
  - ‹+,i›: join of i
  - ‹-,i›: leave of i
  - A view V is more up-to-date than view W if the set of tuples in W is a subset of the set of tuples in V (notation, W C V)

System Properties

- *Storage safety*: the R/W protocols satisfy the safety properties of an atomic R/W register, [Lamport, 1986]
- *Storage liveness:* every R/W operation executed by a correct client eventually complete
- *Reconfiguration - join safety*: if a server *j* installs a view $v : i \in v$, then server *i* has invoked the *join* operation or *i* is member of the initial view
- *Reconfiguration – leave safety*: if a server *j* installs a view $v : i \notin v \wedge (\exists v' : i \in v' \wedge v' \subset v)$, then server *i* has invoked the *leave* operation.
- *Reconfiguration – join liveness*: eventually, operations are enabled at all correct servers that had invoked the *join* operation.
- *Reconfiguration – leave liveness*: eventually, operations are disabled at all correct servers that had invoked the *leave* operation.

View Generators

- Distributed oracles used by servers to generate sequences of views for system reconfigurations
- Each view *v* has an associated *view generator*, distributed implemented by servers in *v*
- Views are generated according to the following properties:
  - *Accuracy*
    - *Strong*: an unique view sequence is generated at all processes
    - *Weak*: different view sequences can be generated at different processes, but one sequence is contained in the other
  - *Termination*: after initialization at process *i*, the view generator eventually generates a new view sequence at *i* (unless it fails)
  - *Non-triviality*: the views in some generated sequence are always up-to-date than its associated view

Perfect View Generators

- *Perfect View Generator* (Strong Accuracy)
  - Needs consensus: which is impossible in asynchronous systems
  - *Protocol main idea*: execute Paxos-like consensus protocol to decide the new view sequence (containing just one view), in partially synchronous systems
  - View generator properties come directly from agreement and termination properties of consensus

Live View Generators

- *Live View Generator* (Weak Accuracy)
  - Does not require consensus, possible in asynschronous systems
  - Different sequences of views may be generated at different servers for updating the same view v.
  - *Protocol main idea*: processes exchange proposals, with the composition of a new sequence for v, until they converge to a new sequence (when a quorum made the same sequence proposal)
  - *Notice that:* any quorum in the system will intersect in at least one correct server
  - *Thus*, two different generated sequences S_1 and S_2 have the following relation: S_1 C S_2 or S_2 C S_1

Live View Generator

Two generated sequences:

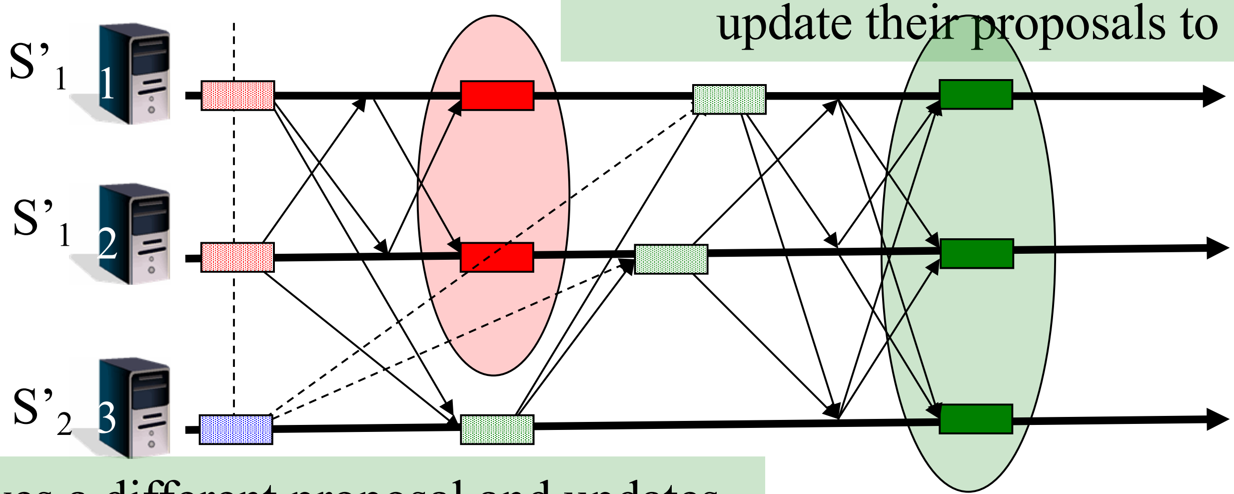
$$S_1 = V_1$$

$$S_2 = V_1 \rightarrow V_2$$

$$S_1 \subset S_2$$

Servers #1 and #2 receive a quorum of equal proposals and generate a sequence $S_1 = S'_1$

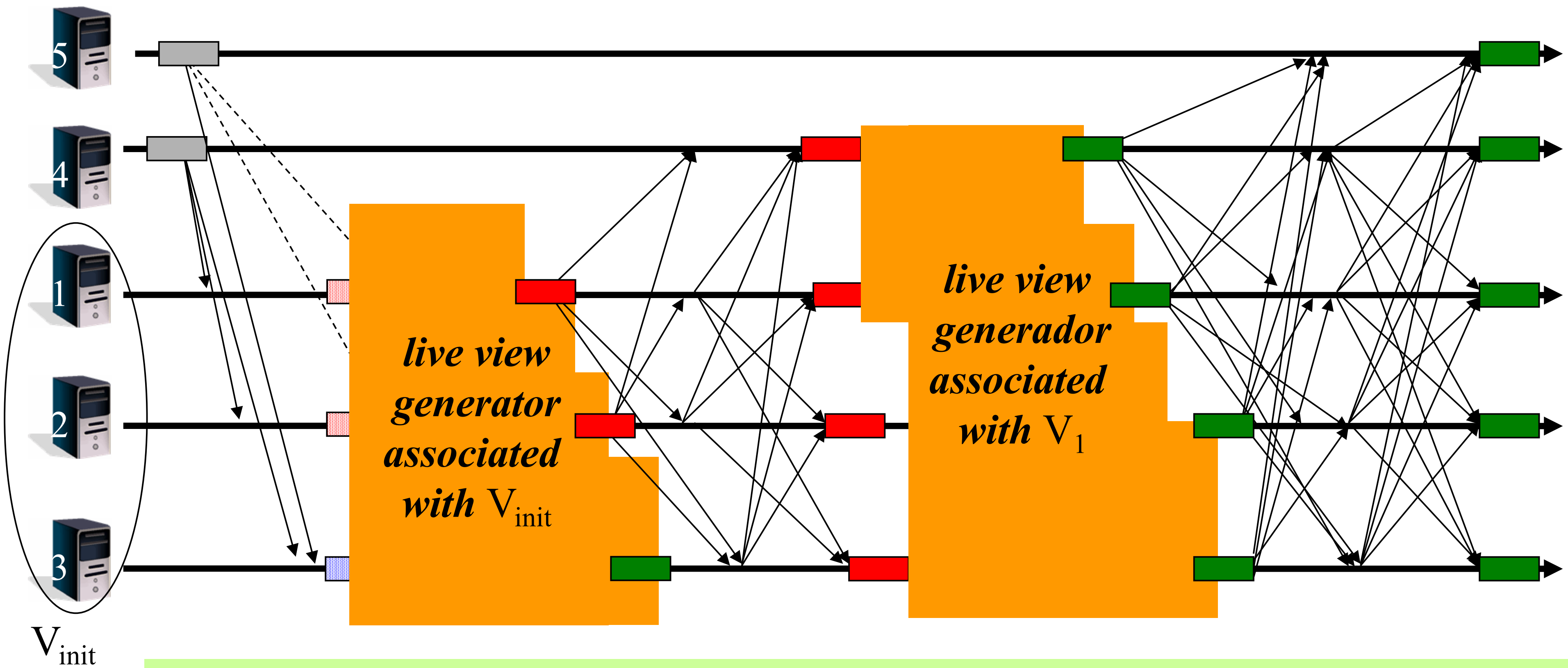
Servers #1 and #2 receive a different proposal and update their proposals to $S'_3 = V_1 \rightarrow V_2$



Server #3 receives a different proposal and updates its proposal to $S'_3 = V_1 \rightarrow V_2$

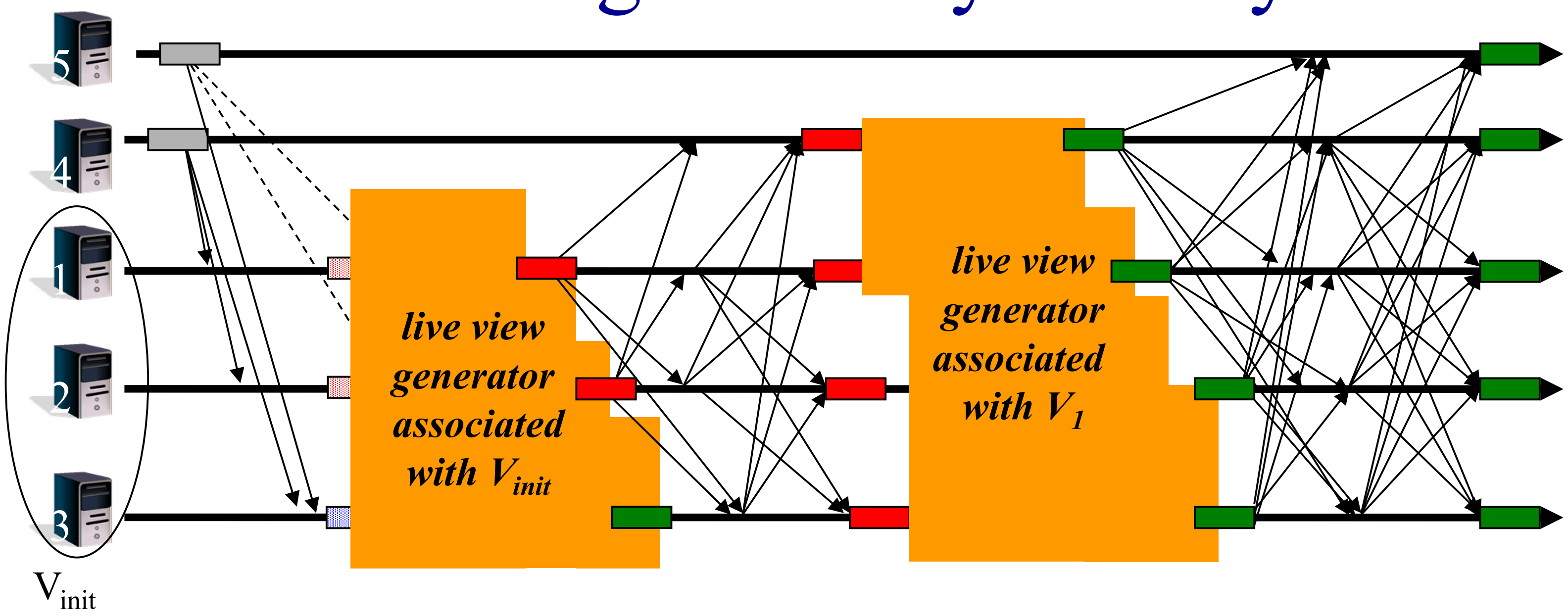
All servers receive a quorum of equal proposals and generate a sequence $S_2 = S'_3$

Reconfiguration using Live View Generators



Notice to deal with asynchrony, processes in V_1 must execute their view generators to obtain a new sequence (even if V_2 is the next view in S_2)

Dealing with Asynchrony



Case 1: processes in V_1 receive S_2, start view generators associated with V_1 and install V_2

$\mathbf{V_1 \longrightarrow V_2}$

Case 2: generators associated with V_1 proposed a sequence $S = W_1$, installed after V_1

$\mathbf{V_1 \longrightarrow W_1}$

Case 3: Cases 1 and 2 concurrently

$\mathbf{V_1 \longrightarrow V_2 \longrightarrow V_2 \cup W_1}$

$\mathbf{V_1 \longrightarrow W_1 \longrightarrow V_2 \cup W_1}$

$\mathbf{V_1 \longrightarrow V_2 \cup W_1}$

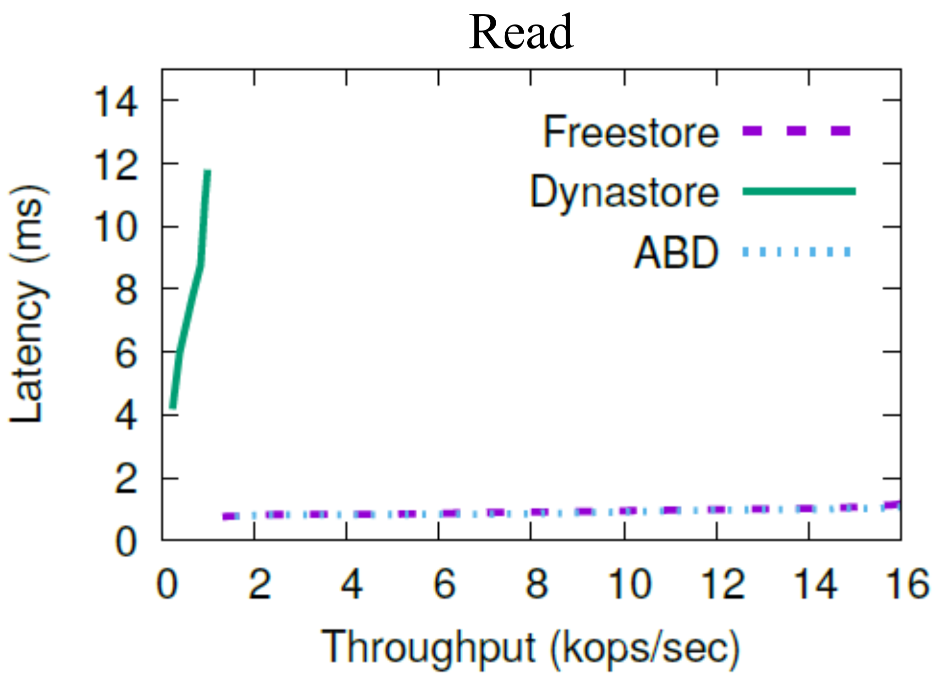
View Installation

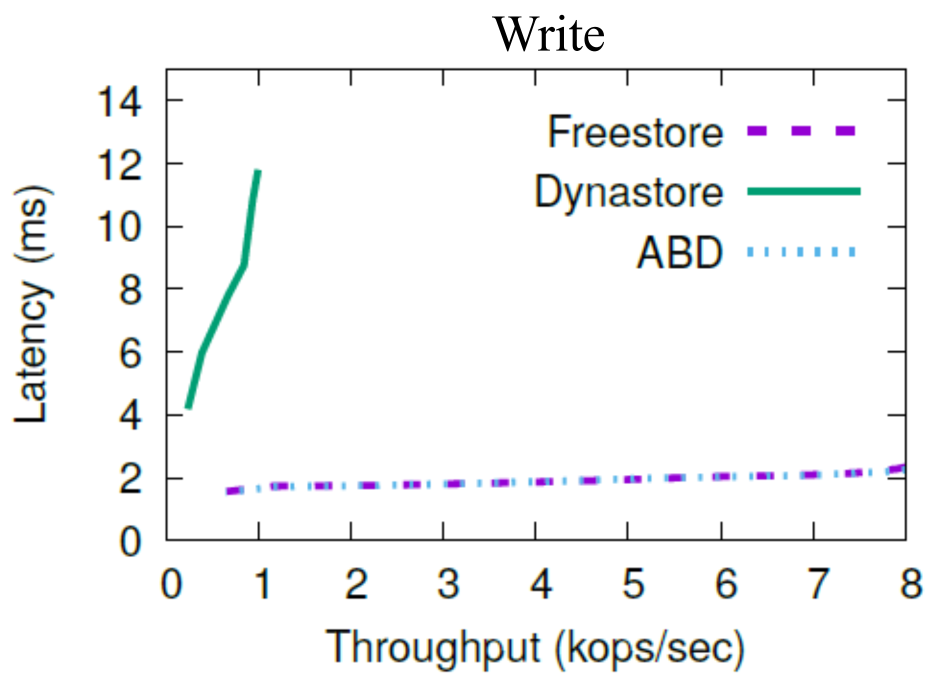
- After a view sequence is generated, these views are "installed" one by one in the system
  - Actually, some views are auxiliary and only the last view in some sequence is installed
- Two messages are necessary: first to inform all processes about a new generated sequence and second for state update
  - During state update, R/W operations are blocked
  - When they are resumed, Clients are directed to the last installed view
  - Using this approach, a Client only need to attach its current view in the R/W protocol messages and servers must verify if it is up-to-date
    - Other approaches need costly access to distributed oracles (implemented by a set of static R/W registers)

Experimental Evaluation

- Two goals:
  - Quantify FreeStore overhead when compared with static ABD
  - Assess the negative impact of a reconfiguration in the performance
- Prototypes of FreeStore, ABD and DynaStore in Go
  - We chose DynaStore to represent existing consensus-free approaches to show that design decisions such as checking a set of static registers to verify if some reconfiguration occurred before executing each R/W, coupling the execution of R/W and reconfigurations, have a significant impact in the performance

Latency vs. Throughput without Reconfiguration



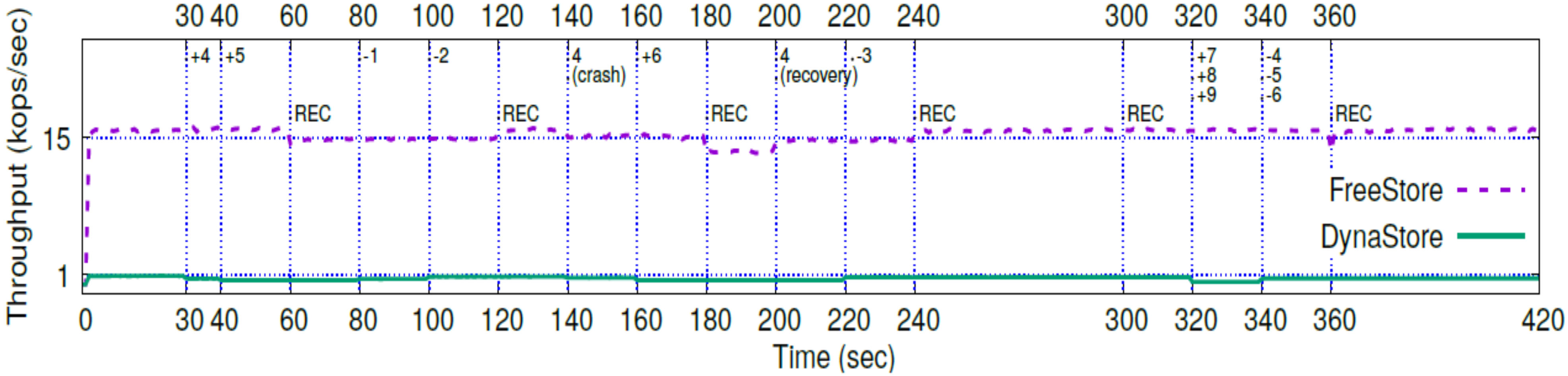


n=3, f=1

18 clients reading/writing a value of 512 bytes

FreeStore imposes a negligible overhead to ABD
(only the current view must be attached in r/w messages)

Reconfigurations and Faults



Initially: n=3, f=1

18 clients reading a value of 512 bytes

FreeStore significantly outperforms DynaStore

- *Mean time for FreeStore reconfiguration: 19ms*
- *r/w blocked in FreeStore for only 4ms*

Conclusion

- FreeStore is a new approach to reconfigure fault-tolerant storage systems, which clarify the differences between relying or not to consensus for reconfiguration
  - It is simpler and require less communications steps than previously proposed solutions
  - It decouples the execution of reconfigurations and R/W algorithms
    - They can execute concurrently, only during state transfers r/w operations are blocked (it is important to notice that in other approaches, R/W operations do not finish before an updated view is installed)
    - A client only need to attach its current view in the r/w messages and servers must verify if it is up-to-date
    - Experiments showed that this approach incorporates a negligible overhead to the static ABD R/W protocol
- Future work: adapt other static R/W protocols to dynamic systems

Thanks!