Reconfigurable Registers
View Synchrony
State-Transfer

DIDA: Class 06

Reconfiguring registers

Reconfiguring Replicated Atomic Storage: A Tutorial

Marcos K. Aguilera* Idit Keidar† Dahlia Malkhi*
Jean-Philippe Martin* Alexander Shraer†1

*Microsoft Research Silicon Valley
†Technion – Israel Institute of Technology

Reconfigurable Registers

Reconfiguring registers

- We have seen (in the distributed systems course at the BsC level) that it is possible to implement atomic registers without consensus (using the ABD algorithm)
- We can ask if it is possible to reconfigure a register system (by removing or adding replicas) without using consensus
- The answer is yes!
- However, it is far from trivial!
  - And I do not have the time to go into this in detail during this course

Reconfiguring registers

- Example: consider a system with configuration config-x = {s1, s2,s3,s4}
- Consider a reconfiguration command R1 to remove s4 and another reconfiguration command R2 to add s5.
- The final result will be config-y = {s1, s2, s3, s5}
- Different nodes may see different intermediate configurations such as {s1,s2,s3} or {s1,s2,s3,s4,s5}
- Assume a client c1 that observes R1 first.
  - It will write on {s1,s2,s3}, for instance, using the majority (s1,s2)
- Assume a client c2 that observes R2 first and reads after the write from c1
  - It will read from a majority of {s1,s2,s3,s4,s5}, say (s3,s4,s5)
  - It will not read the new value!

Reconfiguring registers

- I will only present a simpler form of reconfiguring registers based on some external coordination service, that uses Paxos to totally order all reconfiguration commands.
- This ensures that all replicas of the register (and all clients) see the same sequence of configurations
  - In the previous example, this would prevent client c2 from using configuration {a1,s2,s3,s4,s5}: because R1 happens before R2, the only valid configurations are:

{a1,s2,s3,s4} -> {a1,s2,s3} -> {a1,s2,s3,s5}

Reconfiguring registers

- Unfortunately, even if we totally order all reconfigurations, we can still have problems
- As we have discussed when addressing coordination services, it is possible that different replicas and/or clients are informed of the new configurations with different delays

Reconfiguring registers

- Consider a reconfiguration that moves the system from
  - C1: s1, s2, s3
- To a new configuration (where s4 replaces s3)
  - C2: s1, s2, s4
- A client writes in the new configuration using a majority, say (s2,s4)
- Later another client reads, but still uses the old configuration and reads from (s1,s3); this client fails to read the correct value!

Reconfiguring registers

- If a client *c* knows about C2, before writing in C2, *c* “writes” a pointer to C2 in C1, i.e., it makes sure that a majority of replicas of C1 “know” about C2
- If later another client attempts to read from C1, it will discover about C2 and it will read again from C2
  - Note that when reading from C2, the client can learn about C3, etc.

Reconfiguring Registers

- We have seen that:
  - Even if we have a total order on reconfigurations, consistency is not trivial
  - Because we need to ensure that clients do nt read from the old configuration after the new configuration becomes “active”
- Total order of configurations requires consensus
  - But non-reconfigurable registers do not
- Is it possible to have reconfigurable consensus without consensus
  - Yes but complex
  - Not discussed in this course

View-Synchrony

View-Synchrony

Reliable Communication in the Presence of Failures

KENNETH P. BIRMAN and THOMAS A. JOSEPH
Cornell University

The design and correctness of a communication facility for a distributed computer system are reported on. The facility provides support for *fault-tolerant process groups* in the form of a family of reliable multicast protocols that can be used in both local- and wide-area networks. These protocols attain high levels of concurrency, while respecting application-specific delivery ordering constraints, and have varying cost and performance that depend on the degree of ordering desired. In particular, a protocol that enforces *causal* delivery orderings is introduced and shown to be a valuable alternative to conventional asynchronous communication protocols. The facility also ensures that the processes

View Synchrony

- Warning: you may have seen these slides in the “Distributed Systems Course” @ IST

View Synchrony

- An abstraction to program in a system where the set of servers may change in time
- Was proposed by the team of Prof. Ken Birman from Cornell U.
- Implemented in a series of commercial and open-source products, including ISIS, Horus and Ensemble, among others

View Synchrony

- The system was extremely influential in the area of dependable systems
- A significant limitation was that the design and many of the implementations assumed a perfect failure detector
- Thus, the resulting system could generate unsafe executions in scenarios with network partitions
- One can consider extending view-synchronous systems to work with an imperfect failure-detector but Paxos become the “de-facto” standard on that setting

Regular vs uniform reliable broadcast

- Regular reliable broadcast (supported by most view-synchronous systems):
  - (Regular) agreement: if a correct process delivers a message, all correct processes deliver that message
- Uniform reliable broadcast
  - Uniform agreement: if a process delivers a message, all correct processes deliver a message

- This sounds like a small detail but has big implications

Consider the following scenario

- We have 3 processes, p1, p2, p3 that have received messages *a,b,c* and *d*.
- Process p1 sends and delivers a message *x* and then becomes disconnected from the rest of processes
- Processes p2 and p3, do not deliver *x* and then proceed to send and receive messages *y* and *z*.
- The state of p1 is is the result of processing [*a, b, c, x*]
- The state of p2,p3 is the results of processing [*a, b, c, y, z*]
- The two partitions have diverged!

View synchrony

- Independently of the problems related with the perfect failure detection assumption and the regular/ uniform broadcast choice, the view-synchronous model is interesting because it highlights the challenges in dealing with dynamic configurations

View synchrony

- The system evolves in a (totally ordered) series of views:
  - V1 = {p1, p2, p3, p4, p5, p6}
  - V2 = {p1, p2, p3, p4, p6}
  - V3 = {p1, p3, p4, p6}
  - V4 = {p1, p3, p4, p6, p7}

- Informal: all process that remain in the views, receive the same sequence of views
  - If p1 receives V1, V2, V3, V4 then p3 also receives the same sequence

View Synchrony

- A process that has received view V1 but not view V2 is said to be be "in view V1"
- A process is "correct" in V1 if in was a member of V1, a member of V2, and delivers V2.
- Messages are broadcast "in a view"
- Messages broadcast by a correct process in some view Vx and delivered in Vx to all other processes
- All processes that move from view Vx to view V(x+1) deliver the same set of messages in view Vx

View Synchrony

- Most VS implementations guarantee causal order among messages delivered.
- Total order can be trivially implemented “on top” of VS

The challenges of view change

- Consider that processes are running in V2 = {p1, p2, p3, p4, p6} and that p2 is detected to be failed (remember we are using a perfect failure detector)
- The system should “install” V3 = {p1, p3, p4, p6}
- Assume that p1 keeps sending messages in V2. According to our definition, these messages must be delivered still in V2. This delays the installation of V3 (possibly, for ever).

View-change protocol

- Complex protocol that involves:
  - Temporarily ”stopping” all senders in view Vx
  - Discover which messages different processes may have already delivered in Vx
  - Make sure all processes agree on V(x+1)
  - Make sure all processes in V(x+1) also obtain and deliver those messages
  - Deliver V(x+1) and resume message transmission, now in V(x+1)

- This was known as the “flush” protocol

Coordinator based flush-protocol

- Coordinator starts with an estimate of V(x+1)
- Sends “block and flush” to all processes in V(x+1)
- When another process receives “block and flush” it: i) stops sending more messages in Vx, ii) stops delivering more messages in Vx; iii) sends to the coordinator a list of all messages it has delivered in Vx.
- Coordinator waits for replies from all processes in V(x+1); if one process fails before sending the response, it is excluded from V(x+1)
- Coordinator sends “missing” messages to each process in V(x+1)
- Coordinator instructs all nodes to deliver V(x+1)

Coordinator based flush-protocol

- Things can get really messy if coordinator fails during this process
- New coordinator needs to be elected and start the entire procedure from scratch

Consensus-based view-change

- We can see the view change protocol and a consensus problem where nodes need to agree:
  - Who is in the next view
  - The set of messages to be delivered in the previous view

Consensus-based view-change

- We can replace the coordinator based view change protocol by a protocol based on consensus (such that the failure of the coordinator is resolved “inside” consensus).
- This can be done using an all-to-all communication pattern
  - All nodes ask all other nodes to stop
  - All nodes send the set of messages delivered so far to all other processes
  - Different nodes my have different perspectives of which nodes are live or failed
  - A process may advance to consensus when it has a set V(x+1) such that it has received a reply form all processes in V(x+1) and it considers that other nodes have failed

Consensus-based view-change

- The input to consensus is a “value” in the form:

  <

  V(x+1)={pi, pj, …},

  set-of-messages delivered by processes that belong V(x+1) in Vx

  >

- The output of consensus is a new view and a set of messages that must be delivered before that view

Total-order on top of VS

- When deliver Vx
  - **cleanup ()**
  - sequencer = lowest_id (vx)
  - unordered = empty-set
- When to-broadcast
  - vs-broadcast (<DATA, m>)
- When vs-deliver (<DATA, m>)
  - unordered = unordered + {m}
  - If “I am the sequencer” then
    - vs-broadcast (<ORDER, id(m)>)
- When vs-deliver (<ORDER, id(m)>)
  - unordered = unordered - {m}
  - to-deliver (m)
- **cleanup ()**
  - **??????**

Total-order on top of VS

- When deliver Vx
  - **cleanup ()**
  - sequencer = lowest_id (vx)
  - unordered = empty-set
- When to-broadcast
  - vs-broadcast (<DATA, m>)
- When vs-deliver (<DATA, m>)
  - unordered = unordered + {m}
  - If “I am the sequencer” then
    - vs-broadcast (<ORDER, id(m)>)
- When vs-deliver (<ORDER, id(m)>)
  - unordered = unordered - {m}
  - to-deliver (m)
- **cleanup ()**
  - sort (unordered)
  - to-deliver all messages from unordered in the sorted order

why does this work?

State Transfer

State transfer

- So far I have omitted an important aspect of reconfiguration: state transfer
- When a new node joins a configuration, it needs to obtain the state at the end of the previous configuration before it starts executing request from the new configuration
- For instance, in Paxos, a new node in a configuration needs to learn all the commands executed in previous configurations, before starting to execute commands from the new configuration

State transfer

- State transfer can be performed by transferring the log of commands or by sending the full state of the application
- For instance, if the state machine keeps the balance of a bank account, it may be simpler to transfer to the new machine the value of the account when a old machine is stopped than the entire sequence of deposits and withdrawals
- The new members must keeps the new commands it receives from the new machine in a pending state before it gets the state. Only after, it may process those commands.

State transfer

- In practice the state transfer may impose a significant load on the nodes that send the state, the network, and the node that receives the state
- In some cases, it may be required to halt the state machine for a period to ensure that the state-transfer terminates and the new node can ”catch-up” with the other nodes