TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Programming Distributed Systems

04 Replication

Annette Bieniusa, Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

Motivation

- Replication is a core problem in distributed systems. [2, Sec 15.1-15.3]
- Why do we want to replicate services or data?
  - **Performance**: If there are many clients issuing operations, a single process might not be enough to handle the whole load with adequate response time. Further, keeping data close to clients reduces the network latency when handling requests.
  - **Availability**: Despite server failures and network partitions, clients can still interact with the system (potentially operating with stale / conflicting / ... data).
  - **Fault-tolerance**: Despite faults, the systems behaves correctly; e.g. it does not loose information.
- We can replicate computations and **state** (focus of this lecture)

State Machine Replication

- A process has a state S, and a set of operations $Ops = \{Op_1, Op_2, \ldots\}$ that either return (read, query) or modify (write, update) that state.
- Clients invoke operations from the set Ops over the system.
- The process is replicated, i.e. there are multiple copies / instances of the same process.

Replication Algorithm

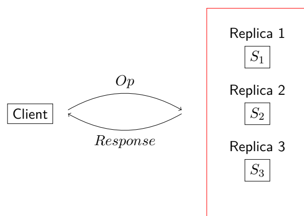
A replication algorithm is responsible for managing the multiple replicas of the process

- under a given fault model
- under a given synchronization model

In essence, the replication algorithm will enforce properties on the effects of operations observed by clients given the evolution of the system (potentially including the evolution the clients).

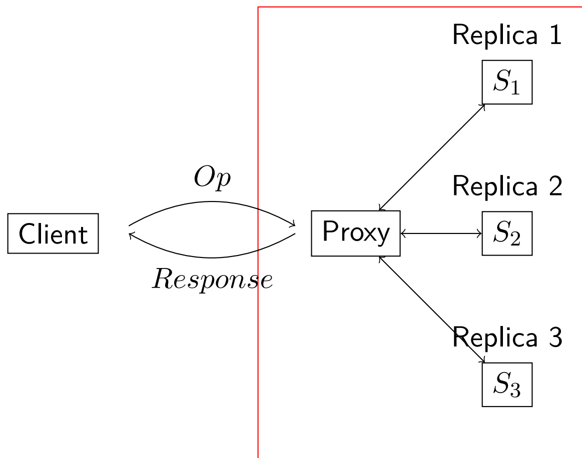
Desirable properties: Transparency + Consistency

- Clients should not be aware that multiple replicas (might) exist.
- When interacting with the system, a client should only observe a single logical state.
- The behavior of this logical state must be in accordance with its correctness specification.

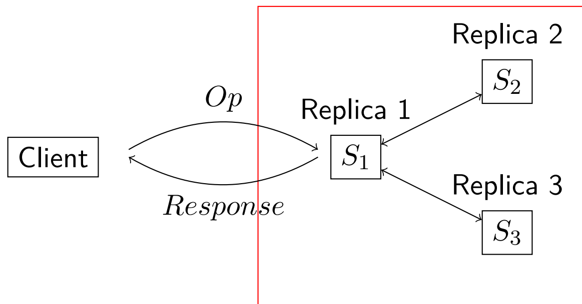


$\Rightarrow$ Need to restrict the state that can be observed by a client!

Solution 1: Coordinating proxy



Solution 2: Only one replica interacts with the client



Replication strategies

- **Active Replication:** Operations are executed by every replica.
- **Passive Replication:** Operations are executed by a single replica, results are shipped to other replicas.

- **Synchronous Replication:** Replication takes place before the client gets a response.
- **Asynchronous Replication:** Replication takes place after the client gets a response.

- **Single-Master** (also known as Master-Slave): A single replica receives operations that modify the state from clients.
- **Multi-Master**: Any replica can process any operation.

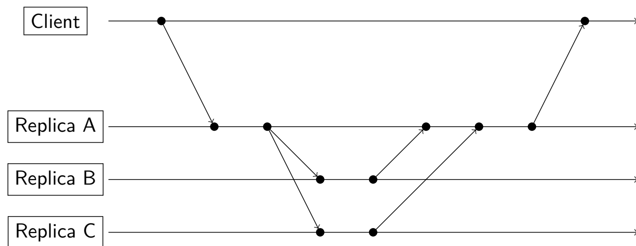
Active Replication

- All replicas execute operations.
- State is continuously updated at every replica $\Rightarrow$ Lower impact of a replica failure
- Can only be used when operations are deterministic (i.e., they do not depend on non-deterministic variables, such as local time, or generating a random value).
- If operations are not commutative (i.e., execution of the same set of operations in different orders lead to different results), then all replicas must agree on the order operations are executed.

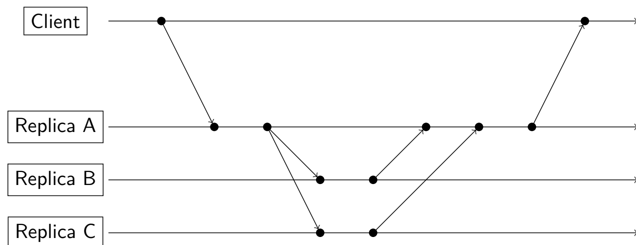
Passive Replication

- Appropriate when operations depend on non-deterministic data or inputs (random number, local replica time, etc.)
- Load across replicas is not balanced.
  - Only one replica effectively executes the (update) operation and computes the result.
  - Other replicas only observe results to update their local state.

Synchronous Replication

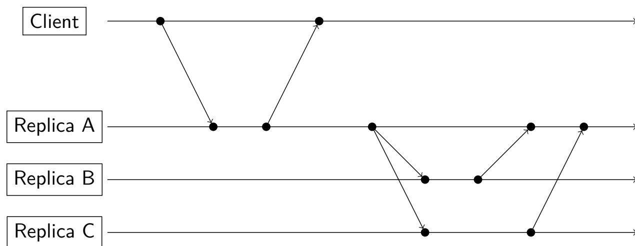


Synchronous Replication

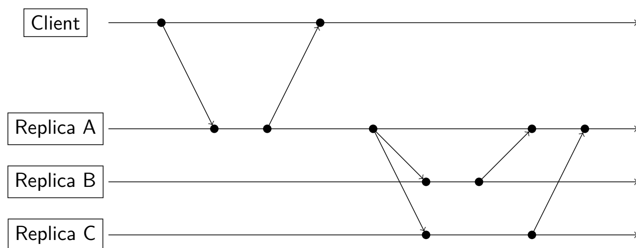


- Strong durability guarantees: Tolerates faults of $N - 1$ servers
- Request will be served as fast as the slowest server
- Response time is further influenced by network latency

Asynchronous replication



Asynchronous replication



- Replica immediately sends back response and propagates the updates later.
- Client does not need to wait.
- Tolerant to network latencies
- *Problem:* Data loss if the replica A goes down before forwarding the update!

Single-copy (Master-slave, Primary-backup, Log Shipping)

- Only a single replica, called the master/leader/coordinator, processes operations that modify the state.
- Other replicas can process client operations that only observe the state (read operations).
- Problems:
  - Clients might observe stale values!
  - Susceptible to lost updates or incorrect updates if nodes fail at inopportune times!
- When the master fails, someone has to take over the role of master.
- If two processes believe themselves to be the master, safety properties might be violated.

Multi-master Systems

- Any replica can process any operation (i.e, both read and update operations).
- All replicas behave in the same way $\Rightarrow$ better load balancing
- *Problem:* Divergence
  - Multiple replicas might attempt to perform conflicting operations at the same time, which requires some form of coordination (e.g. distributed locks or other coordination protocols) that typically are expensive.

Preventing divergence in multi-master systems

- *Idea*: Execute all operations in the same order on all replicas

$\Rightarrow$ **Atomic broadcast** (aka Total order broadcast)

Preventing divergence in multi-master systems

- *Idea*: Execute all operations in the same order on all replicas

⇒ **Atomic broadcast** (aka Total order broadcast)

Properties:

- *Validity*: If a correct process a-broadcasts message m, then it eventually a-delivers m.
- *Agreement*: If a correct process a-delivers message m, then all correct processes eventually a-deliver m.
- *Integrity*: For any message m, every process a-deliveres m at most once, and only if m was previously a-broadcast.
- *Total order*: If some process a-delivers message m before message m', then every process a-delivers m' only after it has a-delivered m.

Implementing Atomic Broadcast

We rely on the **consensus** abstraction to implement atomic broadcast.

- Each process p_i has an initial value v_i ($propose(v_i)$).
- All processors have to agree on common value v that is the initial value of some p_i ($decide(v)$).

Properties of Consensus:

- *Agreement*: Every correct process must agree on the same value.
- *Integrity*: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- *Termination*: All processes eventually reach a decision.
- *Validity*: If all correct processes propose the same value v, then all correct processes decide v.

Atomic Broadcast: Algorithm

```
State:
  k                // consensus number
  delivered        // messages a-delivered by process
  received         // messages received by process

Upon Init do:
  k <- 0;
  delivered <- ∅;
  received <- ∅;
Upon a-Broadcast(m) do
  trigger rb-Broadcast(m);
Upon rb-Deliver(m) do
  if ( m ∉ received ) then received <- received ∪ {m};

Upon received \ delivered ≠ ∅ do
  k <- k + 1;
  undelivered <- received \ delivered;
  propose(k, undelivered);
  wait until decide(k, msg^k)
  ∀ m in msg^k in deterministic order do trigger a-Deliver(m)
  delivered <- delivered ∪ msg^k
```

- Every process executes a sequence of consensus, numbered 1, 2, . . .
- Initial value for each consensus for the process is the set of messages received by p, but not yet a-delivered.
- msg^k is the set of messages decided by consensus numbered k
  - Each process a-delivers the messages in msg^k before the messages in msg^{k+1}
  - More than one message may get a-delivered by one instance of consensus!

Equivalence of Atomic Broadcast and Consensus

- One can build Atomic Broadcast with Consensus.
- One can build Consensus with Atomic Broadcast (how?).

Consensus and Atomic Broadcast are equivalent problems in a system with reliable channels.

Question

How do you solve consensus in

- an asynchronous model
- with crash-fault
- and (at least) one failing process?

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

The FLP Theorem

2001 Dijkstra prize for the most influential paper in distributed computing

Theorem[3]

There is no deterministic protocol that solves consensus in an asynchronous system in which a single process may fail by crashing.

Proof Idea

- *Idea:* We construct a run where
  - at most one process is faulty
  - every message is eventually delivered
  - but no processor eventually decides

We will now present the essential steps in the proof.

FLP: System model

We will use here a slightly different model that simplifies the proof.

- $N \geq 2$ processes which communicate by sending messages
- Message (p, m) where p is receiver and m content of the message
- Message are stored in abstract *message buffer*
  - $send(p, m)$ places message in buffer
  - $receive(p)$ randomly removes a message from buffer and hands it to p or hands "empty message" to p
- This model describes an asynchronous message delivery with arbitrary delay!
- *Requirement:* Every message is eventually delivered (i.e. no message loss)

FLP: Configurations

- A *configuration* is the internal state of all processors + contents of message buffer.
- In each step, a processor p performs a $receive(p)$, updates its state deterministically, and potentially sends messages. We call such a step an *event* e.
- An execution is defined as a (possibly infinite) sequence of events, starting from some initial configuration C.

FLP: Assumptions

- **Termination**: All correct nodes eventually decide.
- **Agreement**: In every config, decided nodes have decided on the same value (here: 0 or 1).
- **Non-triviality** (Weak Validity):
  - There exists one possible input config with outcome decision 0, **and**
  - There exists one possible input config with outcome decision 1
    - For example, input "0,0,1" $\rightarrow$ 0 while "0,1,1" $\rightarrow$ 1
    - Validity implies non-triviality ("0,0,0" must $\rightarrow$ 0 and "1,1,1" must $\rightarrow$ 1)

FLP: Bivalent Configurations

- *0-decided configuration*: A configuration with decide "0" on *some* process
- *1-decided configuration*: A configuration with decide "1" on *some* process

- *0-valent configuration*: A config in which *every* reachable decided configuration is a 0-decide
- *1-valent configuration*: A config in which *every* reachable decided configuration is a 1-decide

- **Bivalent configuration**: A configuration which can reach a 0-decided **and** 1-decided configuration

FLP: Bivalent Initial Configuration

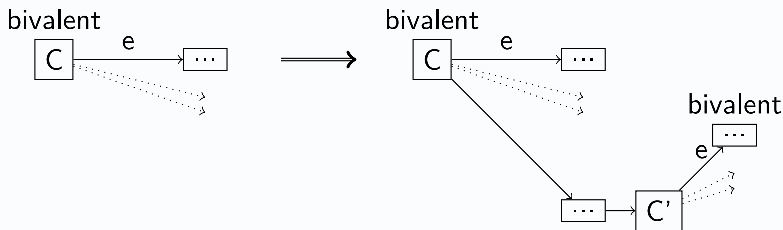
Lemma 1

Any algorithm that solves consensus with at most one faulty process has an initial bivalent configuration.

FLP: Staying Bivalent

Lemma 2

Given *any* bivalent config C and *any* event e applicable in C, there exists a reachable config C' where e is applicable, and $e(C')$ is bivalent.



FLP: Proof of Theorem

1. Start in an initial bivalent config. [This configuration must exist according to Lemma 1.]
2. Given the bivalent config, pick an event e that has been applicable longest.
   - Pick the path which takes the system to another config where e is applicable (might be empty).
   - Apply e, and get a bivalent config [applying Lemma 2].
3. Repeat 2.

Termination violated.

What now?

Impossibility of Consensus is different from the halting problem! Or isn't it?

- In reality, scheduling of processes is rarely done in the most unfavorable way.
- The problem caused by an unfavorable schedule is transient, not permanent.
- Re-formulation of consensus impossibility:

Any algorithm that ensures the safety properties of consensus can be delayed indefinitely during periods with no synchrony. " ##
Circumventing FLP
Obviously, by relaxing the specification of consensus . . .

Impossibility of Consensus is different from the halting problem! Or isn't it?

- In reality, scheduling of processes is rarely done in the most unfavorable way.
- The problem caused by an unfavorable schedule is transient, not permanent.
- Re-formulation of consensus impossibility:

Any algorithm that ensures the safety properties of consensus can be delayed indefinitely during periods with no synchrony. “ ## Circumventing FLP

Obviously, by relaxing the specification of consensus . . .

- *Agreement*: Every correct process must agree on the same value.
- *Integrity*: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- *Termination*: All processes eventually reach a decision.
- *Validity*: If all correct processes propose the same value V, then

Different approaches

- *Idea 1:* Use a probabilistic algorithm that ensures termination with high probability.
- *Idea 2:* Relax on agreement and validity.
- *Idea 3:* Only ensure termination if the system behaves in a synchronous way.

Summary

- Replication is one of the key problems in distributed systems[1].
- Characterization of replication schemes
  - active/passive
  - synchronous/asynchronous
  - single-/multi-master
- Problem: Divergence of replicas
- Atomic Broadcast and Consensus
- FLP Theorem

Next lecture: The Consistency Spectrum

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

[1] Bernadette Charron-Bost, Fernando Pedone und André Schiper, Hrsg. *Replication: Theory and Practice*. Bd. 5959. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-11293-5. DOI: 10.1007/978-3-642-11294-2. URL: https://doi.org/10.1007/978-3-642-11294-2.

[2] George Coulouris u. a. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011.

[3] Michael J. Fischer, Nancy A. Lynch und Mike Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (1985), S. 374–382. DOI: 10.1145/3149.214121. URL: http://doi.acm.org/10.1145/3149.214121.