Reconfiguring a State Machine

L. Lamport, D. Malkhi, and L. Zhou,
ACM SIGACT News (Distributed Computing Column),

Francesco Bongiovanni – INRIA

July 13, 2010
Reconfiguration = changing the set of processes executing a distributed system

The scope of this paper is: \textit{fault-tolerant distributed systems}
Reconfiguration = changing the set of processes executing a distributed system

The scope of this paper is: fault-tolerant distributed systems

The rationale: you can represent any system as a state machine. (well... almost)

The state-machine approach consists of implementing a fault-tolerant DS by:

- describing it as a state machine
- and using a general fault-tolerant algorithm to implement that state machine

---

this feature is one of the contribution of *Time, Clocks and the Ordering of Events in a Distributed System* (1978)
Let the state machine itself specify the configuration

It’s actually an old idea (more than 20 years old)...but it still appears that it’s not well understood.
Important property of a SM: *irrevocability of the output*
Important property of a SM: *irrevocability of the output*

A state machine is a mapping:
\[ \langle \text{Command, OldState} \rangle \rightarrow \langle \text{Response, NewState} \rangle \]
Important property of a SM: *irrevocability of the output*

A state machine is a mapping:

\[
\langle \text{Command, OldState} \rangle \rightarrow \langle \text{Response, NewState} \rangle
\]

A state machine implementation provides an interface by which clients propose commands to the system and receive outputs from it.
Important property of a SM: *irrevocability of the output*

A state machine is a mapping:
\[ \langle \text{Command}, \text{OldState} \rangle \rightarrow \langle \text{Response}, \text{NewState} \rangle \]

A state machine implementation provides an interface by which clients propose commands to the system and receive outputs from it.

*Safety* requirement: outputs received by all the clients are generated by a single sequence of *chosen* commands (each of which has been proposed by a client)
Preliminaries

- Important property of a SM: *irrevocability of the output*
- A state machine is a mapping: 
  \[\langle \text{Command}, \text{OldState} \rangle \rightarrow \langle \text{Response}, \text{NewState} \rangle\]
- A state machine implementation provides an interface by which clients propose commands to the system and receive outputs from it.
- **Safety** requirement: outputs received by all the clients are generated by a single sequence of *chosen* commands (each of which has been proposed by a client)
- **Liveness** property: Mmmmm it depends on how it’s implemented...but generally it goes like this:
  - if enough servers are nonfaulty
  - and eventual partial synchrony is satisfied
  - then proposed commands are added to the sequence of chosen commands and their outputs is delivered to nonfaulty clients.
two sets of servers: acceptors and learners

tolerate $f$ server failures if: at least $2f + 1$ acceptors and $f + 1$ learners
Preliminaries (cont’d)

- two sets of servers: acceptors and learners
- tolerate $f$ server failures if: at least $2f + 1$ acceptors and $f + 1$ learners
- Classical way of implementing the state machine: with a consensus algorithm for choosing a single command
  - the implementation runs a sequence of logically separate instances of the consensus algorithm (instance $i$ to choose the $i^{th}$ command)
  - Most SM implementations use a special subset of learners called leaders, through which a client can propose a command $c$ that will be then assigned number $i$ and proposed as the command to be chosen by consensus instance $i$
  - A leader does not have to wait until command $i$ is chosen to propose another command $i + 1 \Rightarrow$ different commands can be chosen concurrently
Configuration = set of processes (clients, acceptors, ...) executing the system

The consensus algorithm uses a fixed configuration ⇒ we have to ensure that each instance of that algorithm is executed by a single configuration.
Configuration = set of processes (clients, acceptors, ... ) executing the system

The consensus algorithm uses a fixed configuration ⇒ we have to ensure that each instance of that algorithm is executed by a single configuration

Reconfiguration is achieved by using ≠ configurations for ≠ instances

The configuration at cmd $i$ is the one used to execute consensus instance $i$

To prevent chaos, processes must agree on the configuration at cmd $i$
In order to have a reconfigurable SM implementation, we introduce \( \text{cfg} \), a component specifying the current configuration. (i.e. the configuration at cmd \( i \) is determined by the value of \( \text{cfg} \) in the state immediately following execution of command \( i - 1 \))

We add a new cmd: \( \text{rcfg}(C) \), specifying a new configuration \( C \).
Reconfiguration made easy (cont’d)

- In order to have a reconfigurable SM implementation, we introduce $\text{cfg}$, a component specifying the current configuration. (i.e. the configuration at cmd $i$ is determined by the value of $\text{cfg}$ in the state immediately following execution of command $i-1$)

- We add a new cmd: $\text{rcfg}(C)$, specifying a new configuration $C$

- The obvious way to define the SM is to let executing $\text{rcfg}(C)$ in order to set $\text{cfg}$ to $C$. This method is called $R_1$.

- The problem with $R_1$: it prevents concurrent processing of $\text{neq}$ proposed cmds since the configuration used to execute instance $i+1$ of the consensus algorithm can be changed by executing command $i$

  $\Rightarrow$ the SM implementation cannot begin choosing cmd $i+1$ until cmd $i$ has been chosen
To allow concurrent processing of cmds, we define a SM with an additional state such that executing \textit{rcfg}(C) as cmd number \( i \) causes \( cfg \) to change after executing command \( i + \alpha - 1 \), for some positive integer \( \alpha \).

A reconfiguration command can thus take effect \( \alpha \) cmds later, allowing concurrent processing of up to \( \alpha \) cmds. This method is called \( R_\alpha \).
Alternative approach (more intuitive…but way harder)

Reconfiguring the system by:

- Terminating the state machine
- Letting it choose the new configuration
- Resuming the execution with a new state machine using the new configuration
You can implement any (reconfigurable) service using a state machine

you can benefit of the nice properties of a state machine, especially when it comes to verify the correctness of a distributed system when you want to reason about it.


