Building reliable distributed systems

Antonio Paolacci

June 16, 2011

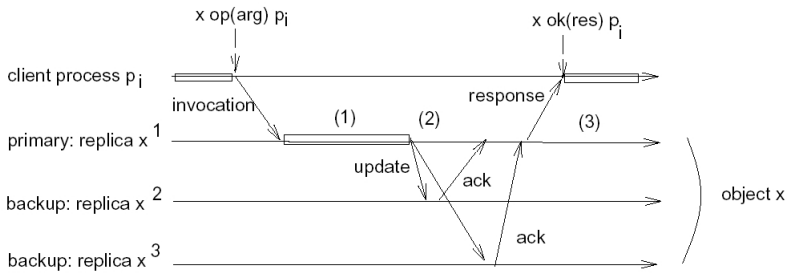
1 Reliable distributed system

Building reliable distributed system is a difficult and ambitious project that faces many issues related to many IT areas. The concept of reliability in a distribuited system could have many facets: a reliable distribuited system have to be *scalable* (the system operate correctly also if it grows in scale for number of users,or number of operative nodes), *fault tolerance* (the ability to recover and continue to operate even with failures nodes without performing incorrect actions), *security* (the ability to preserve data with confidentiality, integrity, availability, authenticity and non-repudiation), *privacy* (the ability to protect the identity and locations of system's users), *timeliness* (the ability to take decisions in a specified time bound), *predictable performance* (to guarantee that the system achieves desired levels of performance), **consistency**(the ability to coordinate actions (read,write operations for example) in a correct and expected manner even in presence of concurrency and failures). We focus on this last one problem.
Consistency criteria: all processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (non unique) global timestamp. To preserving consistency criteria, we have to guarantee the **linearizability**. The linearizability represents a way to execute concurrent operations on shared objects as they were sequential. Naturally two operations OP1 and OP2 with init(OP1) $<$ init(OP2) and return(OP1) $<$ return(OP2) are concurrent iff init(OP2) $<$ return(OP1), or more simply they overlap. Sufficient conditions for implementing linearizable executions of n replicas:

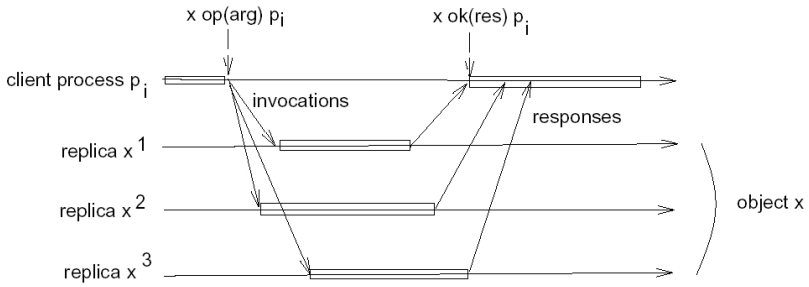
- **Uniform Atomicity**: if a replica completes operation Op then eventually all correct replicas complete Op.
- **Uniform Order**: if two replicas both execute two operation Op and Op^1, then they execute Op and Op^1 in the same order.

To implement linearizability we can use two main techniques: **active replication** or **passive replication**(or primary-backup replication). The system

model considered in both cases is an asynchronous distribuited system with clients,replicas processes and the channels are quasi-reliable(reliable delivery, no creation, no duplication).In passive replication there's a "special replica", the primary, that manages all the request processing. It receive the operation invocation from the client, update all the replicas and when has received all the acks from them, give the response to the client.



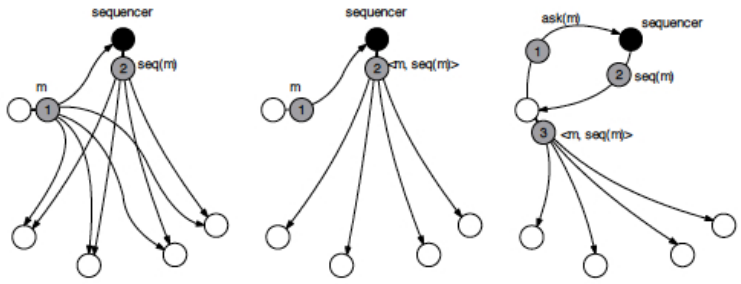
In this model of replication we can have 3 relevant failure scenarios concerning the primary. Faulty backups not represent a problem. The first scenario is that when the failure occurs after the client received the reply. In this case there is no problem, a new primary will be elected. The second scenario is that for which the primary fails before updating any backup. In this case no reply is sent back to clients. Also in this case there's not a true problem, is sufficient that the client re-invokes operation. The third case is the most delicate. In this case, the failure happens between the update for the replicas and the receiving of the acks from the replicas with the problem that some replicas are updated and some others no. To ensure consistency we need **order** and **atomicity**. Order is guaranteed if there's at most one primary that decide enforcing a consistent view of replica group composition. Atomicity is guaranteed if we enforce atomicity of update delivery among backups. To solve the problem we use: **view synchronous multicast(VSM)** as a communication primitive. Views are agreed among processes through another primitive called **group membership service(GM)**. **GM+VSM** solves passive replication. So, summarizing order is guaranteed by the primary, elected using views and the order of members in a view and the update atomicity by the view synchronous multicast.
In the active replication all the replicas are equal. The client send the operation request to all replicas, and wait for the first reply by replicas. Replicas execute requests and give the response to the client.



To restore consistency, when the replicas crash and come back up do a "state-transfer procedure". This procedure is based on an update of the state of the replica that happens thanks to the other alive replicas. To obtain strong consistency is important to ensure:

- **uniform atomicity**: if a replica executes a request eventually all correct replicas execute the request;
- **uniform order**: if two replicas execute the same two request, then they execute them in the same order.

To ensure these proprerties we need to use a **total order multicast** primitive. This primitive can be obtained with a **generic fixed sequencer algorithm**.



(a) Broadcast-Broadcast (b) Send-Broadcast (c) Ask-Broadcast

- Broadcast-Broadcast(BB): the sender broadcast the message m to all the members. Upon receiving m, the sequncer assigns a sequence number seq to m, denoted seq(m) and broadcasts it to all members.
- Send-Broadcast(SB): the sender sends message m to the sequencer, wich assigns a sequence number and then broadcasts the pair $< m, seq(m) >$ to all member.
- Ask-Broadcast(AB): the sender first ask a sequence number to assign m to the sequencer via message-exchange, then the sender broadcasts the pair $< m, seq(m) >$ to all member.

Ordering protocol	Communication primitive	TO specification
Broadcast-broadcast sequencer	$Rcast/Rcast$	$TO(NUA, WNUTO)$
	$URcast/URcast$	$TO(UA, SUTO)$
	$Rcast/URcast$	$TO(NUA, WUTO)$
	$URcast/Rcast$	$TO(UA, WNUTO)$
Send-broadcast sequencer	$Rcast$	$TO(NUA, WNUTO)$
	$URcast$	$TO(UA, SUTO)$
Ask-broadcast sequencer	$Rcast$	$TO(NUA, WUTO)$
	$URcast$	$TO(UA, SUTO)$

This table show which levels of specification(granularity) of total order communication we can achieve in a distributed enviroment with a generic fixed sequencer algorithm. We can conclude analyzing it: all protocols ensure at least TO(NUA,WNUTO) an all protocols employing URCAST ensure TO(UA, SUTO).
Another proposed algorithm solves the consistency problem in active replication shema, because it realizes a TO communication. The following algorithm is called: **privilege-based**. In a privilege-based protocol a single token circulates among processes and grants to its holder the privilege to send messages. Each message is sent with a sequence number derived from a value carried by the logical token. Receiver processes deliver messages according to their sequence number. In several privilege-based protocols processes are organized in a ring and one process pass the token to the next neighboor upon the occurance of an event (no more messages to send, maximum use of token time, maximum of messages sent, maximum use of some resource...).

Privilege-based	$Rcast$	$TO(NUA, WUTO)$
	$URcast$	$TO(UA, SUTO)$

1.1 JGroup

Jgroup helps us to put in practice some of the concepts viewed before, in particular it's a toolkit for reliable multicast communication. With Jgroup we can create groups of processes in which each process can exchange messages with any other process. It supports various operations like: the creation and the deletion of a process, the joining or the leaving of a process in a specific group, the membership detection and the notification when a process,that belong to the group,join the group, leave the group or crash. The communication can occurs between two member of a group, or directly between a member and all the group. When we want to create an application which provides reliable multicast, using jgroup we are sure that reliability will be automatically implemented, so the application don't have to manage the reliability issue. This thing allows developers to gain much time and makes the written code adaptable to different environments.
The system is based on the concept of micro-protocols. For our goal, for

example, JGroup offers two different micro-protocols implementing the total order layer called, **TOTAL**, which embeds an AB fixed sequencer broadcast with Rcast channel, and **TOTAL_TOKEN** which embeds an privilege-based protocol to implement URcast on top of Rcast. These protocol enforce TO(NUA,WUTO) and TO(UA,SUTO) if JGroup is provided with the primary component membership service micro-protocol. JGroups in fact doesn't provide this service natively.

Group Membership Services help us to realize groups communication among application replicas. The role of GMS is to track membership of the groups, a process need to ask to GMS to join the group, drop a process of the group, report a membership changes. Takes as input: process join or leave events, apparent or real failures and output: membership views for group(s) with list of processes, views are used by the application protocol for group(s) communication support. Due to the service itself needs to be fault-tolerant we run multiple istances of GMS and we need to coordinate them. An approch is: to elect the coordinator in the GMS, the oldest of the view as the coordinator or leader. When someone reports to the GMS that a process of the view has failed, the leader run a **2PC protocol** and propose a new GMS view (that exclude the process that has failed) to the other processes. The two-phase commit protocol announces a "proposed new GMS view", excludes the process that has failed, or might add some members who are joining, or could do both at once waits until a majority of members of current view have voted "ok", then commits the change. Now that we have a GMS that reports identical views to all members, we want build a **reliable multicast.** Protocols like TCP, UDP, IDP are not so reliable, some messages might get dropped or some messages could not be sent due to a failure. We have to ensure that all receivers in current view receive any messages that any receiver receives. To do so we introduce the concept of a **"unstable message"**: a message delivered by some processes and some other no. If a process fails we want to "flush" unstable messages out of the system. When a new view is reported all processes echo any unstable messages on all channels on which they haven't received a copy of those messages.Naturally we should first deliver the multicasts to the application layer to achieve a view synchrony (all replicas see the same messages delivered in the same view). For new processes joining the group there's a synchronization mechanism to update their state: when a new view is estabilished,a process already in the group makes a checkpoint and send this information to the new member. If we want a global ordering of messages between the processes, covering also the case of send of concurrent messages by different processes, is not sufficient the FIFO specific.We need of a strong specific, the **fbcast.** Indicating fbcast(a) $->$ fbcast(b) means that we deliver a before b at common destinations. The brother **cbcast** is used in web services where a remote object does a multicas for us, seeming like something we did. So in cbcast

the multicasts have different senders. Other multicast primitives are: **abcast** two messages a and b are delivered in some agreed order at common destinations and **gbcast** a message is delivered like a new group view.

2 The Old World and the New

In the old IT world to guarantee ACID consistency of data was the key. We replicated servers for speed and availability, maintaining consistency. In the new world scalability matters most of all. We have many replicas, but weak consistency among all distributed replicas. In fact is spreading the conviction that consistency is enemy of scalability. Randy Shoup,Werner Vogels,James Hamilton respectively CTO at Ebay,Amazon and Google, refuse to use multicast and are opposed to "consistency mechanisms" because they think consistency-mechanisms are capable of destabilizing the whole data center, doubting of these mechanisms can be scaled adequately. Many communication product are based on IPMC packets (voip,istant messaging...) but IPMC scales poorly! expecially when we have promiscuous routers and NICs $->$ if we use a large numbers of multicast addresses (scalability consequence). System gets larger and more loaded, generalized messages loss starts causing a massive spike of NAK, retransmissions just make things worse, at the end meltdown of IT service! A consistent system try to mantain some properties, but when the network collapses the consistency is considered the fault reason and many now think if we not cared about consistency, instability problems would not have occurred. Consistency $->$ difficult and dangerous!