Autonomic Computer Systems CS321: Group Communication, virt. Synchrony, Paxos and Chubby

September 29, 2011

Prof. Dr. Christian Tschudin
Departement Mathematik und Informatik, Universität Basel

Overview

Context: consensus finding (distributed agreement)

- Remote Procedure Call Semantics
- Introduction Group Communication
- Reliable Broadcast
- Replication
- More on Broadcast: atomic, generic
- Virtual Synchrony
- State of the art: Paxos and Chubby

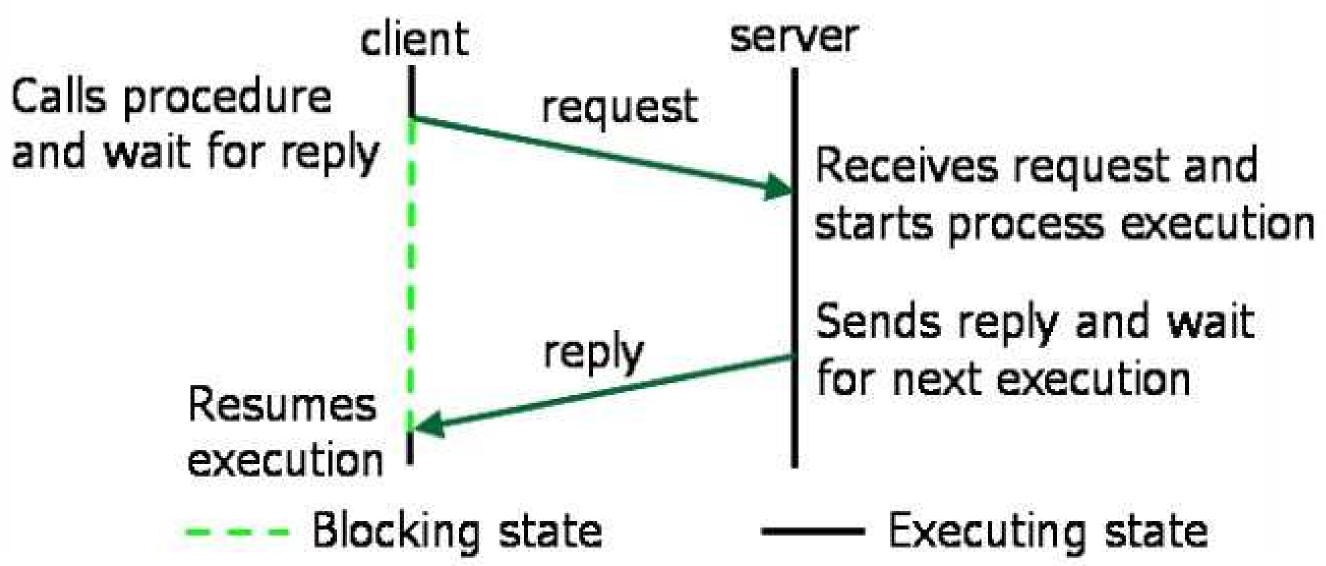
Remote Procedure Call (RPC)

Remote procedure call (Birrel & Nelson 1984)

- Make it appear to the programmer that
  - a normal call is taking place, despite being remote
  - and hide all unnatural `read()`, `write()` primitives
- Process on machine A wants to call procedure on machine B:
  - send details to B
  - process in A is supsended, execution continues at B
  - when process on B returns, control passed back to A

Goal here: show problem of different semantics of RPC.

Remote Procedure Call (2)



Remote Procedure Call (3)

Some issues, not related to the “call (failure) semantics”:

- No (memory access to) global data structures:
  – remote process can only refer to received parameters
  – creates headaches with linked list, for example
- Parameters have to be “serialized”
  – map internal memory representation to data packets (little/big endian), structures etc
  – also called **“marshalling”/“unmarshalling”**
  – how to handle object refs, pointers ?
- Global garbage collection?

Remote Procedure Call (4)

Local procedure calls do not fail, but RPC vulernable:
– network problems, server crash, client crash

- Transparency difficult to maintain:
  – app has to be informed about new kind of errors
- Delivery guarantees:
  – client retransmits requests
  – server filters out duplicates
  – server keeps history of replies, retransmits results

yet, different styles of guarantees. . .

Remote Procedure Call (5)

Semantic of local procedure call is **exactly-once**

- Exactly-once difficult to achieve with RPC
- Instead: **at-least-once**
  – on return from RPC, sure that server saw the request
  – works only if `f(x) = f( f(x) )` (idempotent function)
  – implement: client retransmits request on time-out
- Instead: **at-most-once**
  – on return, RPC called exactly once, *or not at all*
  – implement: server filters duplicate requests
  – still a problem if server crashes during RPC

Remote Procedure Call (6)

Implementation

- RPC involves compiler work:
  – other procedure call details than local call
  – create “stubs”
  – also needs data structure work, marshalling
- Implementations include:
  – SUN RPC (late 1980ies)
  – JAVA RMI (remote method invocation)
  – CORBA middleware

Overview

- Remote Procedure Call Semantics
- **Introduction Group Communication**
- Reliable Broadcast
- Replication
- More on Broadcast: atomic, generic
- Virtual Synchrony
- State of the art: Paxos and Chubby

Group Communication – as middleware

We have datagrams (reliable, unreliable) and remote procedure calls: why yet another type of communication primitives?

- Consensus theory (for example):
  want library of useful distributed services
- Working with sockets and RPC is too low-level
- "Middleware": site between OS and (distributed) application processes and provides highlevel services

Group Communication – issues

Distributed (coordination) services come in many variants:

- Offer different failure semantics and aggregates:
  – send reliably to *all* processes, or discard
  – have all processes receive msgs in *same* order . . .
- Design issues:
  closed/open groups, group membership problem,
  group addressing, atomicity (all-or-nothing),
  msg ordering, scalability.

No "universal" (and final) set of standard groupware services, yet.

Group Communication – "groups"

Example Service: How to agree, inside a defined (distributed) group of processes, on the ordering of events?

- Consensus problem:
  In theory, we cannot solve this problem (asynch!)
- Trick: Modify the group (kick out problematic members),
  such that the remaining members can be made to agree.
- Messages from kicked-out members are then
  ignored, as if these nodes had failed.

Group Communication – “tinker” with group membership

Example: Company has drug X against common cold, which **can** cure people within 24h, but it does not always work.

How to turn it into a fully reliable therapy (great for marketing!)?

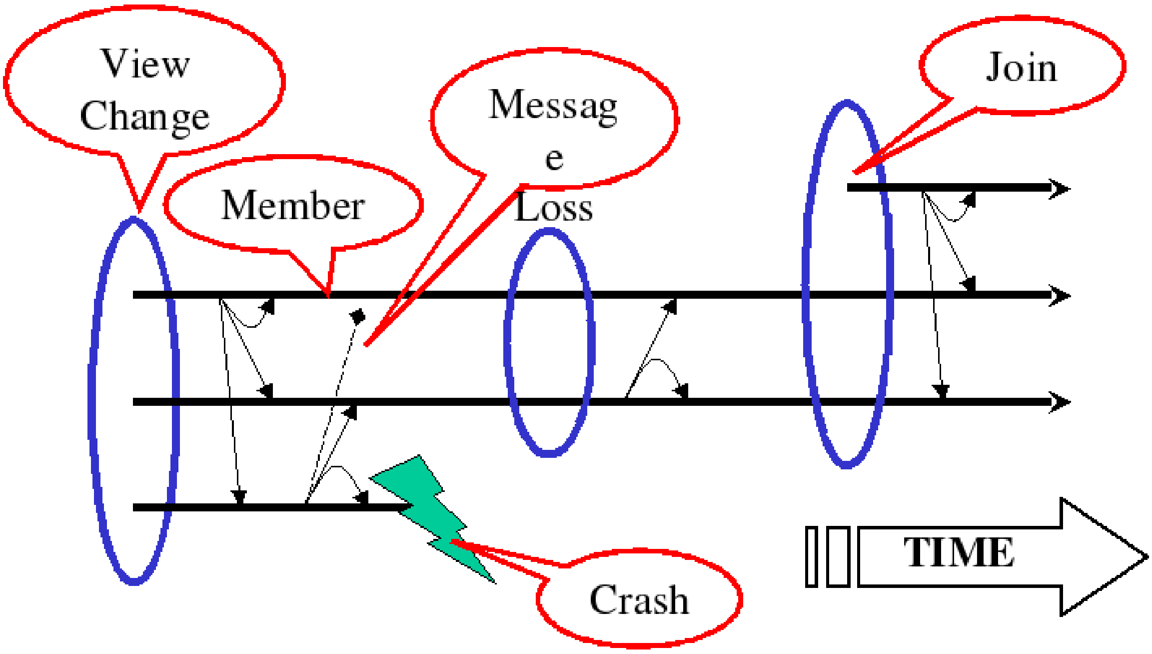
- Add text to the small print: We give guarantee that X works within 24h, should the patient still be alive at that time.
- Implementation: Company observes all patients
  – examines situation at 23h 59’
- If cure did not work, let a hit man “solve” the problem.

Forming Groups

What is a group?

- Explicit membership:
  – nodes have to join
- Primitives:
  – joinGroup(“group”, event-handler)
  – sendPoint2Point(“group”, member-id, message)
  – multicast(“group”, message)
  – leaveGroup(“group”)
- Events can be:
  – ’recv’, ’join’, ’leave’ and ’crash’ events, ’view change’ event

Group Communication: Graphical Representation



(from R. van Renesse)

Overview

- Remote Procedure Call Semantics
- Introduction Group Communication
- **Reliable Broadcast**
- Replication
- More on Broadcast: atomic, generic
- Virtual Synchrony
- State of the art: Paxos and Chubby

Group Communication: Reliable Broadcast

Problem: Implement "all-or-nothing" property for message delivery inside a group, despite message loss.

- "All-or-nothing" property: once a message has been delivered to one destination, it must be delivered to all.
- Useful in databases:
  – send updates to replica servers
  – distributed games: agree how virtual world looks
- Note:
  – all-or-nothing implies consensus
  – **or** reduction of member set

Group Communication: Reliable Broadcast (2)

Protocol (with O(n^2)):
– sender multicasts msg M to all other members
– for every new msg M, each members multicasts it again

- Note: if there are no message losses,
  all nodes receive a copy of M from all other nodes
- Assume node A does not get M, at all:
  – A will not re-broadcast
  – the others will notice, and will send it a copy
- If, after some time, A still did not send M,
  the others agree to kick out A.

Group Communication: Reliable Broadcast (3)

Another protocol, similar to two-phase-commit:

- Sender multicasts “prepare to commit M” message
- all node store M, send back an acknowledgement
- if all acks arrive, senders confirms transaction
- if some ack is missing, sender starts again
- if a process never sends an ack, it is declared dead before restarting the transaction.

Note that it’s a consensus problem, with failure detector!

Other Group Communication Primitives

Seen so far: reliable broadcast. Other possible services:

- FIFO broadcast – msgs from same sender are delivered in order
- Causal broadcast – only causal relationship is preserved
- Barrier – wait until all reached the barrier
- Scatter-gather – send val, all compute, all receive all results.
- Termination detection
- and more . . .

We will still look into: replication, virtual synchrony

Overview

- Remote Procedure Call Semantics
- Introduction Group Communication
- Reliable Broadcast
- **Replication**
- More on Broadcast: atomic, generic
- Virtual Synchrony
- State of the art: Paxos and Chubby

Distributed Algorithms: Replication

Overall assessment of fault tolerance (in distr. systems)

- Transactions, checkpointing:
  – enables recovery from node crashes
  – uses redundancy in time (redo events after recovery)
- Replication:
  – *masks* node crashes
  – use hardware redundancy

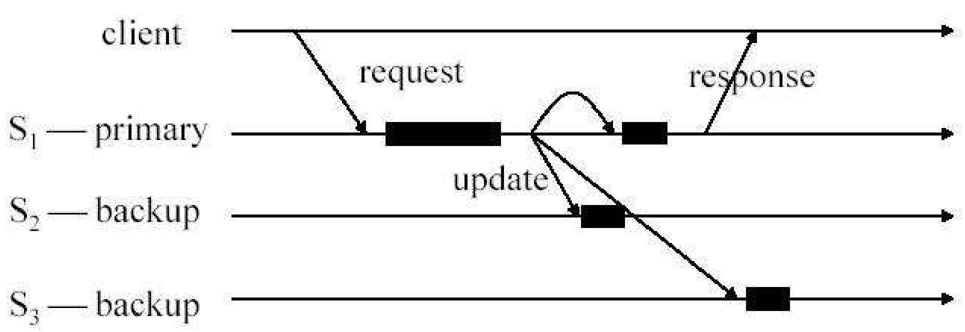
Replication: passive vs active

- passive:
  – one protocol engine, many sites for update log
- active:
  – many protocol engines in parallel

Both approaches have their advantages, disadvantages

Replication: passive

Passive replication, also called *primary backup replication*

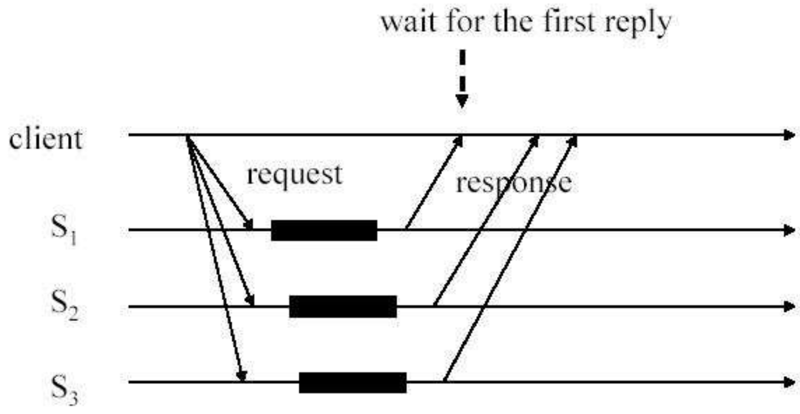


(Fig: Schiper, 2006)

– crash is “decided” based on time-outs
– (we need to address faulty crash decisions)

Replication: active

Active replication: many servers run the (full) protocol



(Fig: Schiper, 2006)

– a client waits for first reply, discards others
– server crash is transparent to clients

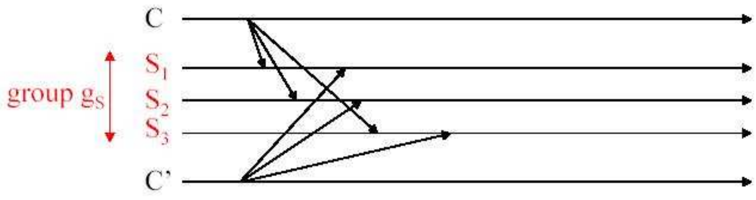
Overview

- Remote Procedure Call Semantics
- Introduction Group Communication
- Reliable Broadcast
- Replication
- **More on Broadcast: atomic, generic**
- Virtual Synchrony
- State of the art: Paxos and Chubby

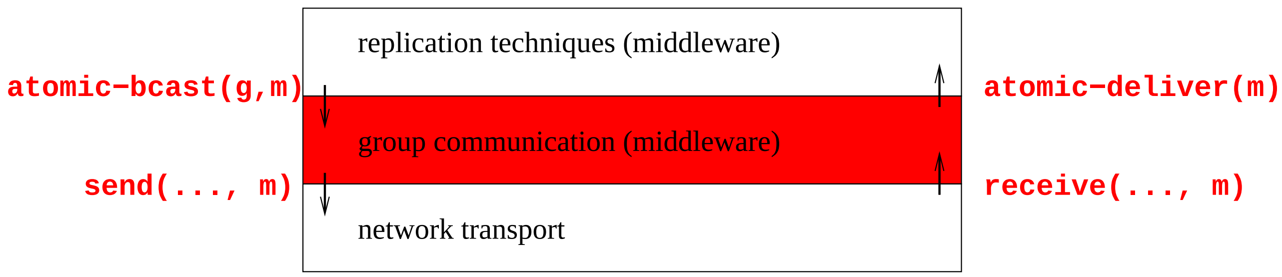
Broadcast primitives (for Replication)

Active replication with many clients: servers must receive requests in the same order (in order to compute the same reponse)

- Useful group communication primitive: **atomic broadcast** (also called total order broadcast)
- The set of replicated servers form a group



Group Communication: (Atomic) Broadcast



- Group communication layer on top of network transport (UDP)
- Group communication provides **atomic or generic broadcast**, possibly using network layer broadcast, or using unicast
- We implement different replication strategies (active, passive) based on atomic/generic broadcast

Atomic Broadcast: Specification

Players: Group g, processes p, q, messages m, m'

- If process p executes atomic-bcast(g,m) and does not crash, then all processes in g eventually do atomic-deliver(m)
- If some process in g does atomic-deliver(m) and does not crash, then all processes in g that do not crash eventually do atomic-deliver(m)
- If two processes p, q in g both do atomic-deliver(m) and atomic-deliver(m'), then they do it in the same order.

Atomic broadcast useful building block for active replication.

Generic Broadcast: Specification

In order to implement *passive* replication:
– we could use atomic broadcast
– but more efficient (and sufficient): generic broadcast

Generic broadcast same spec as atomic broadcast, except that not all messages are ordered.

- Generic broadcast based on a conflict relation on the messages:
  – "conflicting messages" are ordered
  – non conflicting messages are not ordered – not detailed here.

Group Communication: How to handle Crash?

- **"Crash-Stop"** (fail-stop)
  – complete node state is lost after crash (no persistent storage)
  (not realistic: eventually, all nodes are dead)
- Crash-Stop and static/dynamic group:
  – static group: group gets extinct after n crashes
  – even with dynamic group: system not tolerant to catastrophic failure (all nodes crash at the same time)

In practice: Dynamic groups with crash-stop model

Implementing (Group Comm) Broadcast

Atomic as well as generic broadcast contain a consensus problem: Members have to agree on a sequence of message sets, we number these sets.

- Each process p has: counter k, set s of undelivered messages
- Upon `atomic-bcast(m)` do `broadcast(m)`
- Upon `receive(m)` do
  – add m to s
  – **if** no consensus algo running **then**
    `M(k) = consensus(s)`
    foreach m in $M(k)$: `atomic-deliver(m)` in common order
    `k = k+1`

Overview

- Remote Procedure Call Semantics
- Introduction Group Communication
- Reliable Broadcast
- Replication
- More on Broadcast: atomic, generic
- **Virtual Synchrony**
- State of the art: Paxos and Chubby

Group Communication: Virtual Synchrony (VS)

Assessment of group communication primitives seen so far:

- Replication in two forms:
  active (replicated servers)
  passive (one main server, only updates are replicated)
- Two (group comm.) broadcasts:
  atomic (all-or-nothing, ordered)
  generic (as aboce, except ordering only for "conflicting msgs")

Consensus as a possible basis to implement both broadcasts
(agree on ordering of events)

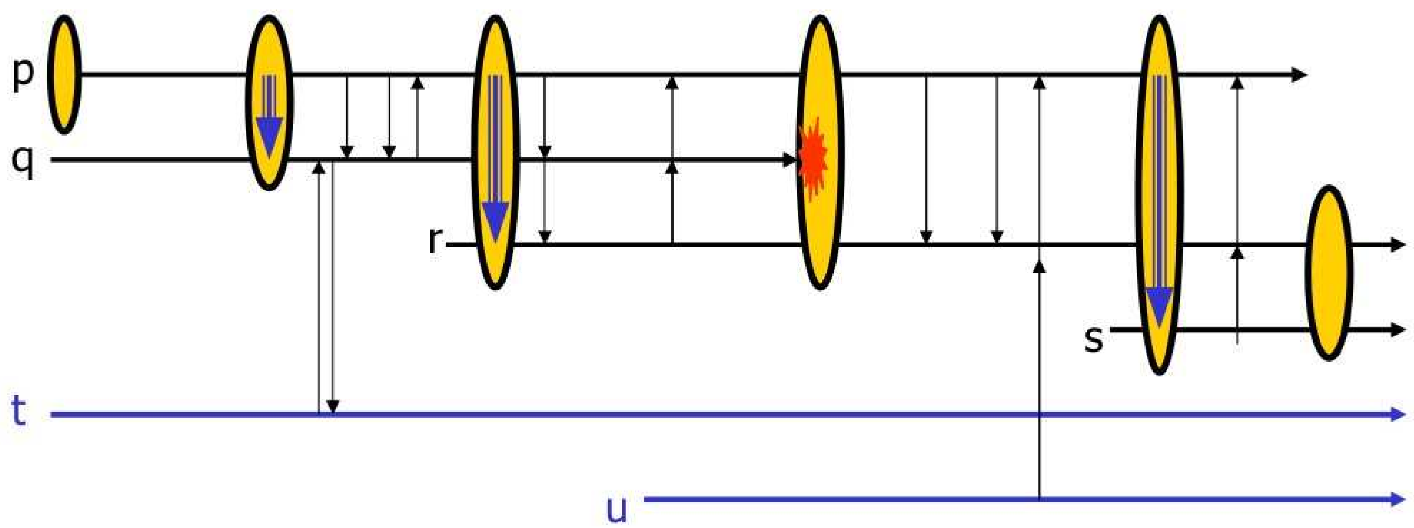
Group Communication: Virtual Synchrony (VS)

Properties of a (pragmatic, when compared to atomic broadcast) “virtual synchrony” service: When receiving a message M,

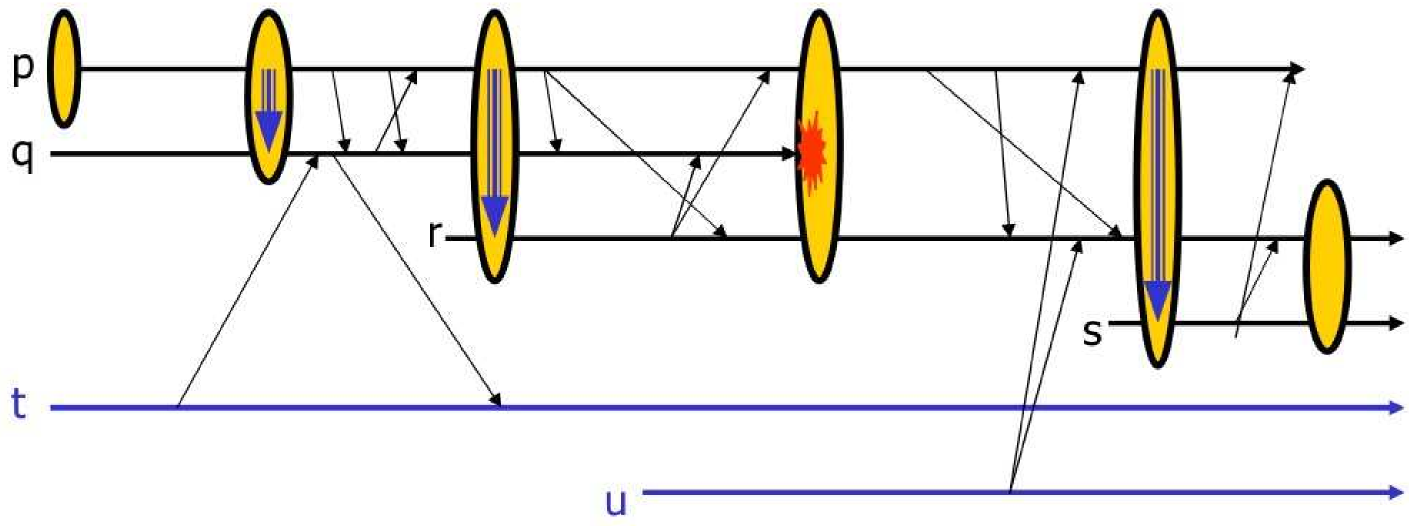
- all receivers have identical group view
- the destination list of M is exactly this view
- all-or-nothing delivery of message M
- ordered delivery (causal or total, see next slide)

Goal: user sees synchronous exec, but it’s asynchronous

“True” Synchrony



Virtual Synchrony



(Difference between Causal vs Total Order)

How should messages be delivered, in group communications?

- Total order:
  All messages multicast to a group are delivered
  to all members of the group in the same order.
  (Possible implementation: central server)

- Causal order (see logical clock discussion):
  Causally related messages from multiple sources are
  delivered in causal order.
  Other messages can be delivered differently, for some members.

Two weaker cases: FIFO (delivery order like sent), or unordered.

Virtual Synchrony (2)

Importance of “views”:

- A message can only be delivered to members in one and the same view.
- Members cannot receive messages sent in a different view than its current view.

Problem to solve, thus: Consensus on views.

Virtual Synchrony – How to solve

Three components needed for VS: have an implementation of

- reliable broadcast (all-or-nothing) (use central server, or acks)
- causal or totally ordered broadcast (use clocks)
- totally ordered membership updates.

VS used in Swiss stock exch, NYSE. Solutions often limited in scale (50-70 members). Solution much easier in ring topology.

Group Communication (end)

- Theory is well advanced on static group and crash-stop model, same for implementation
- Performance an issue, comparing various atomic broadcast algorithms, other set of primitives.
- Still research on-going for other combinations:
  – mostly dynamic group/crash-stop
- In practice: only limited adoption of VS
  (and it's complex: 25'000 lines of code in Ensemble for VS)!

Overview

- Remote Procedure Call Semantics
- Introduction Group Communication
- Reliable Broadcast
- Replication
- More on Broadcast: atomic, generic
- Virtual Synchrony
- **State of the art: Paxos and Chubby**

Weaker Forms of Consistency – Paxos

Virtual synchrony guarantees distributed, fault-tolerant consistency. Can we have more *effective* consensus?

- PAXOS – a family of consensus protocols:
  cheap P., fast P., generalized P., byzantine P., etc
- Three roles:
  proposer, acceptor, learner
- Properties:
  one of the proposed values will be chosen, only one is chosen, only chosen values are learned.

Paxos Algorithm

Works in round with four phases. A single leader called “proposer”:

```
1a: Prepare
  Proposer selects proposal number N, sends a Prepare message
  to a quorum of acceptors.
1b: Promise
  Acceptors refuse proposals with N smaller than last number
  If accepted: sends back last accepted value (for N)
2a: Accept!
  Proposer chooses a value either freely, or if one acceptor already
  accepted a V, then this must be taken. Proposer sends Accept! msg
Phase 2b: Accepted
  If accepetors receive value with correct (promised) number n,
  the value is accepted and sent to all learners.
```

Paxos Algorithm (time seq example, no crash scenario)

```
 Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(N)
   |         |<---------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(N,Vn)
   |         |<---------X--X--X------>|->|  Accepted(N,Vn)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```


Proposer crash: prep msg conflict, new round required

```
Client  Leader         Acceptor     Learner
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(N)
   |      |<------------X--X--X       |  |  Promise(N,{Va,Vb,Vc})
   |      |             |  |  |       |  |  !! Leader fails during broadc
   |      X------------>|  |  |       |  |  Accept!(N,Vn)
   |      !             |  |  |       |  |
   |         |          |  |  |       |  |  !! NEW LEADER:
   |         X--------->|->|->|       |  |  Prepare(N+1)
   |         |<---------X--X--X       |  |  Promise(N+1,{Vn})
   |         X--------->|->|->|       |  |  Accept!(N+1,Vn)
   |         |<---------X--X--X------>|->|  Accepted(N+1,Vn)
   |<---------------------------------X--X  Response
```

Paxos-based Chubby lock service (used by Google)

Chubby: lets distributed clients synchronize *and* agree on basic information:

- whole-write and read of small files,
  advisory locks
  notification events (file modif, node failure etc)
- Chubby is reliable despite node crashes and net partitioning
- easy to use (to program)
- used in GFS to appoint master server, in Bigtable to elect master, discover servers etc

Chubby Services (API)

```
open(), close()
GetContentsAndStat(), GetStat(), ReadDir(),
SetContents(), SetACL(),
Delete(),
Acquire(), TryAcquire(), Release()
GetSequencer(), SetSequencer(), CheckSequencer()
```

Lock related calls: Acquire() etc, as well as GetSequencer() etc

Lock Semantics

- open() in "shared" or "excluse" mode
  (solves "N reader/1 writer" scenario)
- Locks are advisory i.e., read/write do not depend on the lock status
- Sequencer = "snapshot" of lock immediately after the open() call
- A sequencer permits to compare different lock instances, hence reorder messages if necessary
  (by refusing operations in wrong sequence)

Comments on Chubby

- Deliberate merging of:
  – small file system, and name service
  – lock service
  – notifications
- instead of seperate modules (Paxos, etc)
- Scaling: 10-100'000 clients for one Chubby server, re-election within 15 seconds etc
- (reliable) name server is most attractive part

Group Communication (end)

- Theory is well advanced on static group and crash-stop model, same for implementation
- Still research on-going for other combinations:
  – mostly dynamic group/crash-stop
- Performance an issue, comparing various atomic broadcast algorithms, other set of primitives.
- There are ready–made GC libraries (e.g. for Grid applications) to go beyond synchronous point-to-point RPC.
- In practice: weaker forms are sufficient, more scalable