# ***Systems*** Study Notes

Jon Sterling

October 26, 2017

## 1 Implementing RPC

**How does RPC work?** There are several components: user runtime, user stub, server runtime, server stub.

1. Make a local call to the user stub
2. The user stub marshalls this request into appropriate parameters, which it passes to the user RPC runtime instance.
3. The runtime then sends this request along the network to the remote machine, where it is intercepted by the server runtime instance and handed to the server stub
4. The server stub marshalls the request into a local call on the server.

**Advantages and disadvantages**

1. Provides a very nice programming abstraction
2. RPC is completely synchronous, so it may not be right for some applications
3. It is slower than local calls
4. The semantics of exceptions are different from local calls

## 2 Using Threads in Interactive Systems: A Case Study

**Defer work** For forking off jobs, like printing or network requests

**Pumps** For instance, bounded buffers or external devices; ***slack processes*** are an example, which process and consolidate a stream of data to decrease the number of items processed downstream

**Sleepers** Threads which wake up momentarily to perform a single action, e.g. time-outs.

**Deadlock avoiders** In cases where a lock hierarchy can change dynamically depending on runtime conditions, it can be very difficult to release a subset of locks. Sometimes it is easier to release all locks held, and then acquire them all in order where needed. ***Deadlock avoiders*** are a means to accomplish this.

**Task rejuvenators** Threads which monitor processes for errors and restart them.

**Serializers** For instance, a queue and worker.

**Encapsulated fork** A module which implements the general case of a particular use of threads.

## 3 Time, Clocks and the Ordering of Events in a Distributed System

The partial order on events is the least relation closed under the following rules:

$$\frac{P \Vdash a \to b}{a \to b} \qquad \frac{a = P.\mathsf{send}(Q, m) \qquad b = Q.\mathsf{recv}(P, m)}{a \to b} \qquad \frac{a \to b \qquad b \to c}{a \to c}$$

The total order is given by imposing an order on processes. ***The total order is useful for solving the <u>mutual exclusion</u> problem.***

## 4 MapReduce and Spark

MapReduce works in the following way:

1. You define a *map* function which decomposes key-value pairs into sets of intermediate key-value pairs.
2. Then you define a *reduce* function which combines values that share the same key.

Spark aims to extend the MapReduce paradigm to support saving and reusing intermediate data; in MapReduce, to accomplish this you would have to explicitly include a task to write your intermediate data to distributed storage. Spark uses RDDs ("Resiliant Distributed Databases"), which are readonly partitioned data storage, to achieve this.

***This is important for the performance of iterative algorithms.***

## 5 Naiad: A Timely Dataflow System

Naiad differs from MapReduce in that it supports incremental and iterative computation over cyclic data (graphs). This seems important for machine learning.

**Timely dataflow** Structured loops and stateful vertices without global coordination; attaches timestamps in the graph. Vertices are notified once they have received all records for a given round of input or loop iteration.

To achieve ***low latency***, uses asynchronous updates and fine-grained computational dependencies.

## 6 From Shared Virtual Memory to Parameter Servers

**Memory coherence** Reads shall always get the latest-written value.

### 6.1 Centralized Manager

The centralized manager is a data structure which has some procedures that provide mutually exclusive access to some resources. It has a table that maps each page to its ***owning process***, the collection of process which have (read-only) ***copies*** of it, and the ***lock***, which is used for synchronization.

Each process has a `PTable`, with ***access*** and ***lock*** fields for each page.

### 6.2 Distributed Manager

The distributed manager divides page management among the processes. There are several kinds.

**Fixed distributed manager** In this model, each process is assigned a predetermined subset of pages to manage. The problem is that it is difficult to choose this set. ***Better than centralized manager for parallel programs with a lot of page faults.***

**Broadcast distributed manager** In this model, each process manages the pages that it owns. When page-faulting, the process broadcasts into the network to find the real owner of a page. ***Performs badly for parallel programs with high read/write page faults.***

**Dynamic distributed manager** Each process manges the pages that it owns, but it stores a map of the "probable owner" of each page. This obeys the invariant that every chain of probable owners ends with the real owner.

## 7 Paxos

Paxos is based on a two-phase protocol.

1. ***prepare***
    - The proposer sends a `prepare` request with a sequence number $n$ to a majority of acceptors.

- If an acceptor receives a `prepare` request with a sequence number $n$ which is greater than any sequence number it has previously acknowledged, then it promises not to accept any lower sequence number than $n$ ever again, and responds with the highest-numbered proposal that it has so-far accepted.

2. ***accept***

   - If the proposer receives a response from a majority of acceptors, then it sends the `accept` request $(n, v)$ to these acceptors, where $v$ is the value of the highest-numbered proposal among the responses from the acceptors.
   - If an acceptor receives an `accept` request for a proposal numbered $n$, it accepts the proposal unless it has already responded to a proposal with a higher sequence number.

**Multi-paxos** Multi-paxos involves combining multiple instances of paxos to reach consensus on a sequence of values (a log); multi-paxos can be viewed as implementing a distributed state machine. As an optimization, elect a leader rather than having multiple proposers (to avoid competing proposals).

## 8 Dynamo

Dynamo is a distributed key-value store.

**CAP Theorem** Choose two of ***consistency***, ***availability*** and ***partitioning***. Since there will always have to be partitioning, you really have to simply choose between availability and consistency. Dynamo chooses availability and adopts an ***eventual consistency*** model. Conflicts are resolved by application clients.

**Sloppy quorum** In a sloppy quorum, reads and writes are performed on the first $n$ ***healthy*** nodes on the preferred list (rather than the first $n$ nodes). This is done with a ***hinted handoff***, which means that when a node is unhealthy, the message is sent instead to the next healthy mode, but along with a hint that it was meant for a particular node. Then if the unhealthy node comes back to life, the healthy node is supposed to forward the hinted message back to it.

**Consistent hashing** An approach to distributed hash tables which minimizes the amount of remapping when resizing to just $K/n$, where $K$ is the number of keys and $n$ is the number of slots. This is implemented using a ***ring buffer***

## 9 RAID: redundant array of independent disks

**RAID 0** disk striping: no protection

**RAID 1** mirroring: no data loss if drive fails

**RAID 2,3** (bad, nobody uses)

**RAID 4** parity disk: but the dedicated parity disk is a bottleneck

**RAID 5** rotating parity disk: resolve the bottleneck by keeping the parity info in all the disks rather than in a dedicated disk

## 10 LFS: log-structured file system

**inode** locates file blocks, and contains metadata

**inode map** locates inodes in the log

**indirect block** locates blocks of large files

**segment summary** identifies the contents of a segment (file numbres, block offsets)

**segment usage table** counts the live bytes still left in segments, stores last write time for data in segments

**superblock** holds static configuration info such as the number of segments and segment size

**checkpoint region** locates blocks of inode map and segment usage table, identifies last checkpoint in log. ***stored in fixed location, not in log.***

**directory change log** records directory operations to maintain consistency of reference counts in inodes

## 11 AFS: Andrew File System

Scale and performance in a distributed file system.

**Architecture** Two services (client ***Venus***, server ***Vice***). ***Vice*** is a collection of servers where the files reside. ***Venus*** intercepts the filesystem calls and caches files from ***Vices***, storing modified copies of files back in the servers they came from. ***Venus*** only contacts ***Vice*** when a file is opened or closed; reading adn writing individual bytes of a file are performed on the cache, bypassing ***Venus***.

### Key ideas

- whole-file caching, flush on close; when opening cached files, check with ***Vice*** to see if file needs update
- only contact ***Vice*** at open and close, enables scaling
- optimizations: use callbacks instead of checking (polling) to see if file needs update; use file identifier to reduce path traversal overhead.
- consistency model approximately like GitHub.