Consensus and related problems

Today
- Consensus
- Google’s Chubby
- Paxos for Chubby
Consensus and failures

- How to make process agree on a value after one or more have proposed what the value should be?
  - Should we abort or proceed the mission?
  - ...

- ...reach consensus even if some processes may fail
  - Failure – System cannot meet its promises
  - Error – Part of system’s state that can lead to a failure

- Many ways to fail ...
Failure models

- Failures
  - Crash failures – A component simply halts, but behaves correctly before halting
  - Omission failures – ... fails to respond to incoming requests
  - Timing failures – Correct output, but outside a specified real-time interval
  - Response failures – Output is incorrect
  - Arbitrary/byzantine failures – May produce arbitrary output and be subject to arbitrary timing failures

- Basic approach to masking faults – Redundancy
  - Information – Add extra bits to a message for reconstruction
  - Time – Do something more than once if needed
  - Physical – Add copies of software/hardware
Consensus – Definition

- To reach consensus
  - Every process $p_i$ begins in *undecided* state and *proposes* a value $v_i$
  - Processes exchange values
  - Each sets the value of a *decision variable* $d_i$, entering the *decided* state

- Requirements
  - Termination – Eventually all correct processes decide
  - Agreement – All correct processes decide the same
  - Integrity – If the correct processes all proposed the same value, then any correct process in the decided state has chose that value
Dreamland solution – Just as illustration

- A system where processes cannot fail

  Each of $N$ process $p_i$ R-multicasts its proposed value to $g$
  
  Every process $p_j$
  - Collects all $N$ values (including its own)
  - Evaluates $a = \text{majority}(v_i \text{ for } i: 1..N)$, returns $a$ or NIL if no majority exists

- Requirements
  - Termination guaranteed by the reliability of multicast
  - Agreement and integrity by the definition of $\text{majority}$ and the integrity property of reliable multicast
  - (every process receives the same set of values and runs the same function, so they must agree on the same value)
Consensus and related problems

- Besides consensus ...
- Byzantine general problem
  - 3+ generals need to agree to attack or retreat
  - Commander issues an order, lieutenants decide what to do
  - One of the generals may be “treacherous”
  - ~consensus, slightly different integrity: if commander is correct, all decide on the value proposed by commander
- Interactive consistency
  - All processes agree on a vector of values, one per process
  - Similar requirements, e.g. for integrity: if $p_i$ is correct, all correct processes decide on $v_i$ as the $i$-th entry of its vector
- All can be define in the presence of crash or arbitrary failures and for synchronous or asynchronous systems
Consensus in synchronous systems

- Up to $f$ of the $N$ processes exhibit crash failures
- Algorithm proceeds in $f+1$ rounds, in each the correct processes B-multicast the values between themselves
  - Round duration is set based on max time for a correct process to multicast a message

Algorithm for process $p_i$

On initialization: $Values_{i}^{1} := \{v_i\}; Values_{i}^{0} := \{\}$

$Values_{i}^{r}$ holds the proposed values known to $p_i$ at the beginning of round $r$

1. In round $r$ from $[1, f+1]$
2. B-multicast($g, Values_{i}^{r} – Values_{i}^{r-1}$)    \(\text{// send only new values}\)
3. $Values_{i}^{r+1} := Values_{i}^{r}$
4. while (in round $r$) {
5.     On B-deliver($V_j$) from some $p_j$
6.     Insert $V_j$ in $Values_{i}^{r+1}$
7. }
8. After $(f+1)$ rounds
9. $d_i$ is minimum($Values_{i}^{f+1}$)
Consensus in synchronous systems

- Termination is obvious because the system is synchronous (fix number of rounds of fixed length)

- Agreement and integrity holds if all processes arrive at the same set of values at the end of the final round
  - By contradiction, if two processes differ in their final set, some correct process $p_i$ has a value $v$ that another correct process $p_j$ does not
  - Only possible explanation, a process $p_k$ managed to send $v$ to $p_i$, but didn’t get to $p_j$, before crashing
  - But for that, $p_k$ must have received it in a previous round in which $p_j$ didn’t, so there must be a process $p_l$ …
  - And on and on, for each of the $f+1$ round,
  - But we have assumed at most $f$ crashes!
Byzantine general problem

- Three or more generals have to agree to attack/retreat
  - One, the commander, issues the order to \( n-1 \) lieutenants
  - The lieutenants must decide whether to attack or retreat
  - One or more generals, \( m \), may be “treacherous” (faulty)

- If the commander is a traitor
  - Can propose attack to one lieutenant and retreat to another

- If the lieutenant is a traitor,
  - Can tell one lieutenant that the commander said attack and tell another that he said retreat

- A variant of consensus with a distinguished process proposing a value
  - Integrity – if the commander is correct, all correct processes agree with what the commander proposed
Byzantine general problem

- **Assumptions**
  - Synchronous system: Correct processes can detect absence of a message through a timeout (but can't conclude the sender has crashed)
  - Arbitrary failures: A faulty process can send any message with any value at any time (or omit to send)
  - The communication channel between processes are private

### Impossibility with three processes

If $L_1$ has to decide to accept $v$ on the first example, has to choose $w$ in the second one.

- **Lamport et al. solution** solves BGP for $3m+1$ or more generals in the presence of at most $m$ traitors
Algorithm $\text{BG}(0)$

1. The commander ($C$) sends value to every lieutenant
2. Lieutenants use the value received from $C$ or RETREAT if they received no value

Algorithm $\text{BG}(m)$, $m>0$

1. $C$ sends value to every lieutenant
2. For each $i$, let $v_i$ be the value Lieutenant $i$ receives from $C$ or RETREAT. Lieutenant $i$ acts as $C$ in algorithm $\text{BG}(m-1)$ to send the value $v_i$ to each of the $n-2$ other lieutenants
3. For each $i$, and each $j \neq i$, let $v_j$ be the value Lieutenant $i$ received from Lieutenant $j$ in step (2) (using algorithm $\text{BG}(m-1)$), or else RETREAT. Lieutenant $i$ uses the value $\text{majority}(v_1, \ldots, v_{n-1})$
A run with $N \geq 4$ and $f = 1$

- A lieutenant is the traitor
  - L1: $\text{majority}(v,v,x) = v$
  - L2: $\text{majority}(v,v,y) = v$

- The commander is the traitor
  - L1: $\text{majority}(v,w,z) = \text{NIL}$
  - L2: $\text{majority}(v,w,z) = \text{NIL}$
  - L3: $\text{majority}(v,w,z) = \text{NIL}$
Consensus and related problems

- It is possible to derive a solution to one problem using a solution to another

- Suppose there’s a solution to consensus, Byzantine generals and interactive consistency
  - \( C_i(v_1, v_2, \ldots, v_N) \) returns decision value of \( p_i \) in a run of the solution to consensus
  - \( BG_i(j, v) \) returns decision value of \( p_i \) in a run of the solution to Byzantine generals where commander \( p_j \), proposed value \( v \)
  - \( IC_i(v_1, v_2, \ldots, v_N)[j] \) returns the \( jth \) value in decision vector of \( p_i \) in a run of the solution to interactive consistency

- IC from BG – by running BG \( N \) times, one per process
  - \( IC_i(v_1, v_2, \ldots, v_N)[j] = BG_i(j, v_j) \) (\( i, j = 1 \ldots N \))

- C from IC
  - \( C_i(v_1, v_2, \ldots, v_N) = \text{majority}(IC_i(v_1, v_2, \ldots, v_N)[j]) \)

- …
We discussed solutions for consensus and Byzantine generals problem
  - Our solutions assume synchronous system

What if the system is asynchronous?
  - No algorithm can guarantee to reach consensus with even one faulty process [Fischer et. al, 1985]
  - Basic idea: there’s always a continuation of a processes’ execution that avoids reaching consensus (why? Can’t tell if a process is running slow or is dead)
  - Guarantee – it is possible …

How do we work around this?
  - Masking faults: process keep data in persistent storage so that they can restart after crashing (Chubby!)
  - Using perfect by design failure detectors – processes agree to a maximum response time (otherwise, the process has failed)
  - …
Back in 5’
Google’s Chubby

- A fault-tolerant system that provides a distributed locking mechanism and stores small files
- A multi-faceted service providing
  - Coarse-grained distributed locks
  - A file-system offering reliable storage of small files
  - Can be used to support election of a primary in a set of replicas
  - Used by Google FS and Bigtable
  - Also used as a name service within Google
- *That diverse?* Not really, one core service – a practical solution to distributed consensus
Chubby architecture

- A Chubby instance – cell
  - Typically one per data center
  - Replication for fault-tolerance – few replicas (5), with a master
  - Replicas are placed in failure-independent sites to avoid correlated failures
  - … have access to persistent storage that survives crashes
  - … each maintains a small DB whose elements are entities in the Chubby namespace

- Applications access replicas via a Chubby library
Chubby architecture

- **Session and caching**
  - A session – a relation between a client and a cell
  - Maintained using a *KeepAlive* handshake
  - Client caching of data, metadata and info on open handles
    - Unlike GFS, small files are accessed repeatedly
  - Consistency – A mutating operation (e.g., *SetContents*) is blocked until all associated caches are invalidated
    - Invalidation piggybacked onto *KeepAlive* msgs
    - Cached data never updated directly

- **Determinism** promoted the use of it as name service (DNS allows naming data to become inconsistent)
Chubby interface

- Abstraction based on a file system
- Files are organized into a hierarchical namespace
  
  /ls/chubby_cell/directory_name/…/file_name

- As a FS, whole-file operations
  - Reads/writes of files are atomic operations

<table>
<thead>
<tr>
<th>Role</th>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Open</td>
<td>Open a file/directory, returns a handle</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td>Close the file associated with the handle</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td></td>
</tr>
<tr>
<td>File</td>
<td>GetContentAndStat</td>
<td>Returns atomically whole content and attributes</td>
</tr>
<tr>
<td>File</td>
<td>GetStat</td>
<td></td>
</tr>
<tr>
<td>File</td>
<td>ReadDir</td>
<td></td>
</tr>
<tr>
<td>File</td>
<td>SetContents</td>
<td>Write the whole content of a file (atomically)</td>
</tr>
<tr>
<td></td>
<td>SetACL</td>
<td>Access control lists</td>
</tr>
</tbody>
</table>
Chubby interface

- As a lock-management tool
  - Locks are advisory
  - Checking for conflicts is on the programmer

<table>
<thead>
<tr>
<th>Role</th>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>Acquire</td>
<td>Acquire a lock on a file</td>
</tr>
<tr>
<td></td>
<td>TryAcquire</td>
<td>Tries to acquire a lock on a file</td>
</tr>
<tr>
<td></td>
<td>Release</td>
<td>Releases a lock</td>
</tr>
</tbody>
</table>

- Simple support for an event mechanism
  - Clients can register for events as an option in the open call
    - E.g., file has been modified
  - Notifications delivered through callbacks
Chubby interface

- As a fault tolerant log
  - All candidates attempt to acquire a lock associated with an election
  - Only one succeeds, the primary
  - Primary writes its id in the associated file
  - Others can so learn the id of the primary

- To support primary election

Once the submitted value has enter the fault-tolerant log …
Chubby and Paxos

- Chubby’s consistency using Lamport’s Paxos
  - Based on maintaining operation logs
  - As logs become long with time, snapshots

- Paxos – a family of protocols for distributed consensus
  - Replica servers may operate at arbitrary speed and may fail and later recover
  - Replica servers have access to stable, persistent storage
  - Messages may be lost, reordered, duplicated; delivered w/o corruption but may take an arbitrarily long time to arrive

- … distributed consensus for asynchronous systems
  - Impossibility result – Paxos workaround it by ensuring correctness but not liveness (i.e., it is not guaranteed to terminate)
Paxos

- Replicas can submit a value to achieve consensus on
- Agreement = all replicas having this value as the next entry in their update logs
- Algorithm is guaranteed to eventually achieve consensus if majority of replicas run long enough with sufficient network stability
  - Paxos-L1 (progress): if there exists a stable majority set of replicas, then if a replica in the set initiates an update, then some member of the set eventually executes the update
  - Paxos-L2 (eventual replication): if replica $s$ executes an update and there is a set of replicas containing $s$ and $r$, and a time after which the set does not experience any communication or process failures, then $r$ eventually executes the update
Paxos algorithm

1. Elect a replica to be the coordinator
2. Coordinator selects a value and broadcast it to all replicas in a msg called *accept*
   - Other replicas either *acknowledge* it or *reject* it
3. Once a majority *acks*, consensus has been reached
   - Coordinator broadcasts a *commit* msg to notify replicas

Assuming there is only one coordinator and no failures
Paxos algorithm – Multiple coordinators?

- Since coordinators can fail, multiple ones can coexist
- And each can select a different value
- For consensus Paxos introduces two mechanisms
  - Assigning ordering to the successive coordinators
  - Restricting each coordinator's choice in selecting a value
Paxos algorithm – Electing a coordinator

- To identify the right one, ordering with an increasing sequence #
  - Each replica keeps the highest seq # seen
  - If bidding as coordinator, pick a unique #, higher than any it has seen, and broadcast it in a *propose* msg
  - If others have not seen a higher bidder, reply with *promise* msg indicating they will reject later coordinators claims with lower # or send a *negative ack* (not voting for you!)
    - *Promise* msg also contains the most recent value heard by sender as a proposal or consensus (or null)
  - If majority of *promise* msgs are received, replica is elected coordinator (majority is known as quorum)
Paxos algorithm – electing a coordinator

- To ensure the propose seq # is unique
  - Pick smallest seq # s that is > any seq # see so that
    \( s \mod n = i_r \) (replicas unique ids are \( i_r \), between 0...n-1)
  - E.g., if the number of replicas (n) is 5, replica id (\(i_r\)) is 3, and
    the last seq # was 20, replica will pick seq # 23 for next bid
Paxos – Seeking and achieving consensus

- Elected coordinator selects a value and sends an *accept* msg with this value to the associated quorum.

- If any of the *promise* msgs contained a value,
  - Coordinator must pick a value from those received
    - The one of the most recent coordinator
  - Otherwise it is free to select its own value
Paxos – Seeking and achieving consensus

- Any member of the quorum that receives the *accept* msg must accept the value and *ack* the acceptance.
- Coordinator waits, possible indefinitely, for a majority to *ack* the *accept* msg.
- If the majority aacks, we have reached consensus
  - Coordinator broadcasts a *commit* msg to notify replicas of this.
  - Else coordinator abandons the proposal and start again.
Multi-Paxos

- In Chubby, a need to agree on a sequence of values
- So algorithm must repeat to agree on a set of entries in the log
  - Log records all Paxos actions – log before sending each msg
    - propose, promise, accept, ack, commit and flush to disk
      - With close-by machines, write to disk is $$$
    - Certain optimizations are possible such as electing a coordinator for a while
      - And avoid *propose* messages
- Any replica can still try to become the coordinator
Summary

- How to make process agree on a value after one or more have proposed what the value should be?
  - Consensus
- In synchronous and asynchronous systems
- Not just theory – Chubby!