Advanced Topics in Computer Systems, CS262B

Prof Eric A. Brewer

Practical Byzantine Fault Tolerance

March 11, 2004

[updated 3/12/04]

I. Motivation

We need to make systems work without having trust all of the components. We call faults with arbitrary, potentially malicious behavior "Byzantine" faults, after the paper by Lamport.

Byzantium was essentially the eastern Roman empire (at the time Constantinople and now Istanbul), which outlived classic Rome by 1000 years. They spoke Greek and lived in (modern) Turkey, but were Roman in culture. The pejorative adjective "byzantine" comes from a few ideas: a) that their culture was super bureaucratic and complicated, b) that their literature was "uncultured" compared to Classic Greek (think soap operas and intrigue), and c) that they were devious amoral people/traders. From Webster:

> ***4:*** *(often not capitalized) a: of, relating to, or characterized by a devious and usually surreptitious manner of operation <a Byzantine power struggle>*
> *b: intricately involved*

This paper is important because it got the system community take Byzantine fault tolerance seriously, by connecting to a real problem (NFS) and making the performance acceptable and the assumptions reasonable (no synchrony in operations).

II. Background

Synchronous vs. Asynchronous systems

- Synchronous: nodes take steps in synchrony -- the system proceeds in "rounds" and everyone agrees on what round it is, and messages stay within their round
- Async: steps occur at different times and speeds and different nodes. Harder to understand, but more realistic.
- using only read/write shared memory, you can't reach consensus on an async system with faults (can't distinguish faults from delays). Can easily solve with some timing info or atomic read-modify-write operations

Linearizability:

- sequential consistency = one-copy serializability
- linearizability = sequential consistently + respect real-time ordering of events
- Bayou is sequentially consistent, but not linearizable, since a disconnected node may have updates that commit much later than their real-time order.

Fault model:

- "fail stop" means that faulty nodes just stop, which means if you are collecting response

you just need one (you don't get *bad* responses)

- "Byzantine" means the faulty nodes can do anything, and if you collect votes you need f+1 that are the same. Note: this guarantees the consensus value, not the correct value. Correctness thus really means that the non-faulty nodes agree, not the they're "right"

Safety and Liveness:

- safety: bad things don't happen (very specific proof for each desired property)
  - here safety means linearizability
  - safety does not cover clients, instead we limit their damage via access control
- liveness: some good thing eventually happens (can't delay it forever)
  - uses some timing bounds to ensure eventual progress (partially synchronous)

III. The Protocol

Build a replicated state machine that is linearizable; must agree on the operations and their order

Basic idea: need to hear from enough nodes with the same value to ensure that the non-faulty nodes agree

- Assume 3f+1 nodes, with at most f faults
- Assume signed messages
- Assume deterministic behavior
- Assume no systematic failures (like bugs!) -- technically this would be more than f faults

Basic claim: if A hears 2f+1 responses with the same value x, and B hears 2f+1 responses with the same value y, then x = y. (counts their own responses)

- At least f+1 of those response must be from non-faulty nodes
- With only 3f+1 total nodes, there must be at least one non-fault node in common, therefore x = y from that node and thus overall

Basic version:

- client sends request to the primary
- primary assigns order and broadcasts it to the others
- replica execute the operation and reply to client
- client waits for f+1 replies with the *same* value, (at least one which is non faulty)

Request: m=<REQ, o, t, c>c o=operation, t=local timestamp, c = client id, signed by client

Reply: <REPLY, v, t, c, i, r>i

- v = view number, t = client's timestamp, c = client id, i = replica number, r = result, signed by replica i
- Clients waits for f+1 valid replies:
  - valid signatures

- f+1 different values of i
- same t and r for all f+1 messages

Pre-prepare phase: (primary -> replicas)

- primary sends <PRE, v, n, d>p [signed by primary]
  - v=view, n=sequence number (this is the commit order), d=digest of m
  - this is the proposed commit order for this operation
  - primary also broadcasts m, but may do so using a different communication primitive
- a collection of f+1 of these messages received by different replicas is proof that this operation received this sequence number
- Backup accepts the PRE message if:
  - signature is correct and d matches m
  - backup is in view v
  - has not already accepted a different message for operation n

Enter *prepare* phase

- broadcast <PREPARE, v, n, d, i>i to all replicas
- log both PRE and PREPARE messages (so you won't forget -- exactly once semantics)
- add others PREPARE messages to your log if: signatures, n, d match
- prepared(m, v, n, i) iff:
  - i has logged PRE message and 2f matching PREPARE messages (from others)
  - this means that i agrees with the eventual commit order! (others will agree later if they do not already)

Enter commit phase

- enter when you are prepared().
- multicast <COMMIT, v, n, d, i>i
- accept others correct COMMIT messages
- committed-local(m, v, n, i) iff
  - prepared(m, v, n, i) is true, and
  - received 2f+1 matching commits (counting i's)
  - this implies that the system will eventually commit this message
- committed(m, v, n) is true if committed-local() is true in f+1 non-faulty replicas

Garbage collection:

- idea: periodically sign digests of the state
- 2f+1 such digests form a proof that we agree on the state at that time
- can then throw away logs for older state

IV. View Changes

idea: must change the primary if we aren't making progress

- this is just for *liveness* and not safety
- primary = v mod |R| -- so primary rotates with every view change
  - faulty nodes can be the primary for at most f view changes
  - primary selection is deterministic, which simplifies selection
- views essentially overlap in time
  - every operation that is in progress in the old view, we need to re-establish in the new view
  - implies that we must ***re-issue*** the preprepare and prepares messages for operations in progress! (with the new view number)

Backups detect the stall and initiate a view change

- stops making progress, just works on view changes (and checkpoints)
- <VIEW-CHANGE, v+1, n, C, P, i>i
  - n is sequence number for the last stable checkpoint, s, known to i
  - C is the proof of s, i.e. a list of 2f+1 signed checkpoint messages
  - P is a set of sets, P_m, that are proofs that the operation for message m is prepared, which is just the list of 1 pre-prepare message and 2f matching prepare messages for m. We need one P_m for every message that is in progress (prepared but not committed).

When the new primary (the one for view v+1) receives 2f valid view change requests:

- broadcast <NEWVIEW, v+1, V, O>p
- V is the set of 2f view-change requests (a proof of the group desire for a new view)
- O is a set of reissued pre-prepare messages, so that pending operations can get consensus in the new view
- Primary is now in view v+1

Backups move to v+1

- NEWVIEW was properly signed by the new primary
- V is correct
- O is correct, which it can verify by recomputing it locally
- Then sends out PREPARE messages for everything in O
- now in v+1
- May need to get some missing messages; can get the message from any node and then verify it with the digest...