In the previous lecture, we talked about Dissemination quorum system. Today we talk about Phalanx, which uses DQS, coupled with authenticated signature, to model the Byzantine failure model. Remember that the ABD algorithm, discussed in lectures 17 and 18, assumed classical quorum system.

1 Phalanx

Phalanx differs from ABD in utilization of digital signatures by writers (a writer adds its own signature to the write messages sent). More precisely, Phalanx differs from ABD algorithm in the following ways.

1. It uses digital signatures for message authentication.
2. It is based on Dissemination Quorum Systems (DQS).
3. It is wait-free if a quorum in DQS exists that contains only correct processors.
4. Writers are not Byzantine, though writers can fail by crashing.
5. Readers may be Byzantine.

Algorithm 1 serveri code

1: receive \( (\text{read}, cnt) \) from client \( C \)
2: \quad send \( (\text{readack}, val_i, tag_i, sig_i, cnt) \) to \( C \)
3: receive \( (\text{write}, val, ts', cid', sig', cnt) \) from \( C \)
4: \quad val \( (C',(val',ts',C'),sig') \)
5: \quad if \( ts' > tag_i.ts \) or \( ts' = tag_i.ts \land cid' > tag_i.cid \), then
6: \quad \quad \quad \quad \quad tag_i \leftarrow (ts',cid')
7: \quad \quad \quad \quad \quad val_i \leftarrow val'
8: \quad \quad \quad \quad \quad sig_i \leftarrow sig'
9: \quad \quad \quad \quad \quad send \( (\text{writeack},cnt) \) to \( C \)

The server code for a process \( i \) is given in Algorithm 1. Note that the server can be Byzantine.
When a server $s_i$ receives a write message from writer $C$, $s_i$ verifies the signature $sig$ before acting on the message. If the signature verification succeeds and the value of the write is newer than server’s copy, a correct server stores the value, timestamp, and signature of the writer. A correct server uses the locally stored signature to send $readack$ to a read or write client. The write or read clients validate the $readack$ message. This allows values forged by Byzantine servers to be discarded. Since reader never use their own signatures, Byzantine readers can be tolerated.

**Algorithm 2** client$_c$ code for write

1: $inv_c (Phalanx, write(v))$
2: $opCnt ← opCnt + 1$
3: **Phase 1: Timestamp synchronization**
4: send $(read, opCnt)$ to all servers
5: when receive $(readack, val_i, (ts_i, cid_i), sig_i, opCnt)$ and $V(c_i, (val_i, ts_i, cid_i), sig_i)$ from all $s_i$ from some $Q ∈ DQS$
6: $maxts ←$ maximum $ts_i$ in received $readack$ messages.
7: **Phase 2: Write back**
8: send $(write, v, (maxts + 1), σ_c(v, maxts + 1, c), opCnt)$ to all servers
9: when receive $(writeack, opCnt)$ from all $s_i$ from some $Q ∈ DQS$
10: return resp$_c (Phalanx, ack)$

The client side code for write is given in Algorithm 2, and for read in Algorithm 3.

**Algorithm 3** client$_c$ code for read

1: $inv_c (Phalanx, read())$
2: $opCnt ← opCnt + 1$
3: **Phase 1: Read**
4: send $(read, opCnt)$ to all servers
5: when receive $(readack, val_i, (ts_i, cid_i), sig_i, opCnt)$ and $V(c_i, (val_i, ts_i, cid_i), sig_i)$ from all $s_i$ from some $Q ∈ DQS$
6: $maxts ←$ maximum $ts_i$ in received $readack$ messages
7: $maxcid ←$ maximum $cid_i$ in received $readack$ messages with $maxts$
8: $maxval ← val_i$ in received $readack$ message with $(maxts, maxcid)$
9: $maxsig ← sig_i$ in received $readack$ message with $(maxts, maxcid)$
10: **Phase 2: Write back**
11: send $(write, maxval, (maxts, maxcid), maxsig, opCnt)$ to all servers
12: when receive $(writeack, opCnt)$ from all $s_i$ from some $Q ∈ DQS$
13: return resp$_c (Phalanx, maxval)$
2 Consensus Emulation by Classical Quorum

First, we review some background material from lectures 15 and 16.

2.1 Preliminaries on Consensus

Remember that every execution (possibly partial) of an algorithm, which tries to solve the consensus problem, need to satisfy the following three conditions:

1. Validity: Agreed value comes from a process’s initial value.
2. Agreement: All “correct” processes have “same” value.
3. Termination: All “correct” processes terminate with a decision.

Since consistency is the trademark of quorum system, we will focus only on the safety (consistency) conditions while implementing consensus using quorum systems. For that, we partition the set of processes into three non-disjoint categories:

1. Proposers: Propose values that the learners will eventually “learn”.
2. Learners: Agree on a value eventually.
3. Acceptors: Help learners to agree on a value.

Then in this framework, we redefine the consensus requirements in the following way:

1. Validity: If a benign learner “learns” a value ‘v’ and proposers are benign, then some proposer proposed ‘v’.
2. Agreement: No two benign learners learn different values.
3. Termination: If a “correct” proposer proposes a value, then “eventually” every correct learner learns a value.

2.2 Synod consensus algorithm

Assume the asynchronous message passing crash failure model and an eventual leader failure detector Ω (as defined in lecture 16), which applies to a set of proposers. Suppose P different proposers take ids \{1, 2, \ldots, P\}. We also assume a quorum system QS over a set of acceptors that satisfy

\[ \exists Q \in QS, Q \subseteq alive(acceptors). \]
Together, they satisfy the above 3 conditions.

Synod consensus algorithm builds on the Paxos algorithm by Lamport, which is widely used in real life. Synos is a variant of consensus that bypasses the FLP impossibility result, mentioned in lecture 16. So this is known as *Obstruction free consensus (OFC)*. Like consensus, OFC may return ‘learned’ value to a learner, but unlike consensus, OFC can return a special “abort” signal to a proposer. Further, a proposer can propose a value multiple times.

The validity and agreement are same for OFC, as of consensus. But the *termination condition* is different. If a correct proposer $P$ proposes a value, then eventually every correct learner learns a value or $P$ aborts. Also, if a single correct proposer $P$ proposes a value infinitely often, then $P$ does not abort forever.