1 Introduction

1.1 Topics

Rough outline of the first part:

1. Secret sharing
2. Distributed/threshold cryptosystems
3. Asynchronous Byzantine agreement using randomization and using eventual synchrony
4. Atomic broadcast (Byzantine-fault tolerance, BFT)
5. BFT services and storage
6. Proactive cryptosystems
7. Untrusted storage

The second part of the course will be a seminar-style interactive presentation of classic research papers and recently developed systems by the participants.

1.2 Distributed storage tolerating Byzantine faults

1.2.1 Definitions

Byzantine quorum system. Quorum systems are a fundamental concept for synchronizing access to replicated data. Quorum systems usually address systems where servers are subject to crash failures.

Consider a set of \( n \) servers \( \mathcal{P} = \{ P_1, \ldots, P_n \} \), of which up to \( f \) may deviate arbitrarily from their specification, i.e., a system of \( n \) servers with \( f \) Byzantine faults. A Byzantine quorum system for \( \mathcal{P} \) is a set of subsets of \( \mathcal{P} \) such that every two subsets intersect in at least one non-faulty server. Every such set is called a [Byzantine] quorum [MR98].

The canonical example of a Byzantine quorum system treats all servers uniformly and is based on a majority: its quorums are all sets \( Q \subset \mathcal{P} \) such that \( |Q| = \lceil \frac{n+f+1}{2} \rceil \).

Abstract storage. A read/write register is a simple and useful abstraction for shared data storage. Registers were formalized by Lamport [Lam86] in the so-called shared-memory model, where multiple processes access data objects concurrently and asynchronously [AW04]. We use the register abstraction to define the interaction of multiple clients with a storage device connected to clients over a network.
Definition 1 (Register). A register \( r \) is accessed by two operations:

- \( \text{write}(r, x) \rightarrow \text{OK} \): writes a value \( x \) to register \( r \) and returns the symbol \( \text{OK} \);
- \( \text{read}(r) \rightarrow x \): reads the register \( r \) and returns its value \( x \).

A register is characterized along three dimensions:

1. the domain of values that it stores;
2. the number of processes that may write to or read from it; and
3. its behavior under concurrent access.

We consider here only one register; it has arbitrary domain (equivalently, its domain is the set of strings) and it can be accessed by a single writer process and by many reader processes, a so-called SWMR register.

We now address the behavior of registers under concurrent access. Every process executes at any time only one operation. An operation is invoked at some point in time and returns at a later point in time. When a write operation with value \( x \) returns \( \text{OK} \), we say that it writes \( x \).

The sequential specification of a register requires that each read operation returns the value written by the most recent preceding write operation.

For two operations \( o_1 \) and \( o_2 \), we say that:

- \( o_1 \) precedes \( o_2 \) whenever \( o_1 \) returns before \( o_2 \) is invoked (they are sequential), and
- \( o_1 \) is concurrent with \( o_2 \) when neither operation precedes the other one.

Lamport [Lam86] has introduced the following three semantics of a register under concurrent access. W.l.o.g. assume there is an initial write operation that writes \( \perp \).

Safe: A register is safe when every read not concurrent with a write returns the most recently written value. Reads that are concurrent with at least one write may return any value in the domain.

Regular: A register is regular if it is safe and any read concurrent with a write returns either the most recently written value or a concurrently written value.

Atomic: A register is atomic whenever the read and write operations are linearizable [HW90], which means that there exists an equivalent totally ordered sequential execution of them. In other words, there exists a permutation \( \pi \) of all invocations and responses in the execution such that the sequential specification of every register holds and such that for any two operations \( o_1 \) and \( o_2 \) where \( o_1 \) precedes \( o_2 \) in the execution, \( o_1 \) also precedes \( o_2 \) in \( \pi \).

(For one writer only, a simpler definition is to require that the register is regular and ensures that if an operation \( r_1 \) returns a value written by \( w_1 \), an operation \( r_2 \) returns a value written by \( w_2 \), and \( r_1 \) precedes \( r_2 \), then \( w_2 \) does not precede \( w_1 \).)
1.2.2 Distributed implementation of regular storage

Suppose there are is a writer process $C_w$ and multiple reader processes $C_1, C_2, \ldots$; they are collectively called clients. A register accessed by the clients can be implemented in a fault-tolerant way on a distributed system, consisting of $n$ storage servers or replicas, $P_1, \ldots P_n$. Up to $f$ servers may fail by behaving in arbitrary ways (Byzantine faults). We assume that clients do not fail.

The servers communicate with the reader and writer processes by sending messages over an asynchronous network. The network provides a reliable and authenticated point-to-point FIFO channel between every client and every server. The servers do not communicate with each other.

We present a protocol that emulates a register to the reader and to the writer processes, despite the failure of some servers. For tolerating faults, the value in the register is stored collectively by all servers. A wait-free protocol here means that clients complete all operations independently from server failures and independently of the speed of other clients.

The writer may use a digital signature scheme to sign messages, which uses two operations, $\text{sign}$ and $\text{verify}$. The first operation can only be run by the writer $C_w$; calling $\text{sign}_w(m)$ with $m \in \{0, 1\}^*$ returns a signature $\sigma \in \{0, 1\}^*$. The second operation can be run by all clients; $\text{verify}_w(m, \sigma)$ takes $m, \sigma \in \{0, 1\}^*$ as inputs and returns a Boolean value $b \in \{\text{FALSE}, \text{TRUE}\}$ such that $\text{verify}_w(m, \sigma) = \text{TRUE}$ if and only if $\sigma$ was returned to $C_w$ by $\text{sign}_w(m)$ before.

Algorithm 2 (Distributed implementation of a SWMR regular register [MR98]).

Algorithm for the clients. The writer $C_w$ stores a timestamp $t$.

\begin{verbatim}
write$(x)$:  // writer $C_w$ only
    \begin{verbatim}
        t ← t + 1
        σ ← $\text{sign}_w(t\|x)$
        send message (WRITE, $t$, $x$, σ) to $P_1, \ldots, P_n$
    \end{verbatim}
    \begin{verbatim}
    wait for a message (ACK) from $\lceil \frac{n+f+1}{2} \rceil$ servers
    \end{verbatim}
    return OK
\end{verbatim}

\begin{verbatim}
read():  // client $C_j$
    \begin{verbatim}
        send message (READ) to $P_1, \ldots, P_n$
    \end{verbatim}
    \begin{verbatim}
    wait for messages (VALUE, $t_i$, $x_i$, $σ_i$) such that $\text{verify}(t_i\|x_i, σ_i) = \text{TRUE}$
        from $\lceil \frac{n+f+1}{2} \rceil$ servers
    \end{verbatim}
    \begin{verbatim}
    let $x$ be the value $x_i$ received in the message with the largest timestamp $t_i$
    \end{verbatim}
    return $x$
\end{verbatim}

Algorithm for the servers. Every server $P_i$ stores a tuple $(t_i, x_i, σ_i)$.

\begin{verbatim}
upon receiving message (WRITE, $t$, $x$, σ) from $C_w$:  // server $P_i$
    if $t > t_i$ then
        $(t_i, x_i, σ_i) ← (t, x, σ)$
    send message (ACK) to $C_w$
\end{verbatim}

\begin{verbatim}
upon receiving message (READ) from $C_j$:  // server $P_i$
    send (VALUE, $t_i$, $x_i$, $σ_i$) to $C_j$
\end{verbatim}
Theorem 3. Assuming at most $f < n/3$ faulty servers, Algorithm 2 implements a SWMR regular register.

Proof sketch. The read and the write operations each access a Byzantine quorum of servers. Hence, after a write operation terminates, every server in some Byzantine quorum stores the value and the highest timestamp so far. If no other write starts, then at least one server in the Byzantine quorum accessed by the reader will send the written value with the highest timestamp so far in its VALUE message to the reader, and the read returns the most recently written value. If another write operation is concurrent to the read, the unforgeability of digital signatures implies that the reader returns either the most recently written value or the concurrently written value.

References


