Contents

1 General

- Deadline Handin: Friday 23.59
- TA classes start at 10.00
- The book pages is +4

2 Introduction (1)

2.1 IT-Security

2.1.1 Basic Concepts

- There are three fundamental aspects when talking about it security sometimes referred to as *security objectives*
  - **Confidentiality:** we want systems where information does not leak to people that should not have access
  - **Authenticity:** data in systems should be authentic
    - i.e. it has not be tampered with by people who are not authorized
    - Sometimes called *data integrity* when talking about real data
  - **Availability:** systems should work the way they are supposed to
    - i.e. legal users of the systems should be able to get the data when they need them
- The security objectives are intended as loose descriptions of basic user demands to a system

2.1.2 Security Policies

- A **security policy** is a precise description of the security objectives
  - Needs to be much more precise than just talking about the three security objectives
  - In a formal abstract model it can take the form of specifying certain states of the system that are unsafe and should be avoided while others are defined secure
  - In a programming language environment e.g. Java it can take the form of a set of rules for what different programs and software components are allowed to do
    - What data they can access etc.
  - When systems policies involve human beings they are often less precise
- High-level procedures are thought of as part of the security policy while lower technical solutions are referred to as **security mechanisms**

2.1.3 Threat model

- A **Threat model** defines which attacks we are going to worry about
  - Needed because it is not feasible to design as system secure against any attack

2.1.4 Security Mechanisms

- A **Security Mechanism** is needed to ensure that a system follows a security policy when subjected to an attack identified in the Threat Model
  - A security mechanism can use any technique to help us reach our goal
    - Such as physical locks, id cards, password protection, virus checkers, cryptography etc.

2.2 Cryptography

2.2.1 Confidentiality versus Authenticity

- The security objective sought with a cryptographic solution may be either *confidentiality* or *authenticity*
- Cryptology is most often not useful in trying to ensure availability
  - This problem is more depended on the on the way the data is handled and stored physically
- The two kinds of security are very different and must therefore be handled in two different way
- Thus, in a given real-life scenario, you must always find out first whether you are dealing with an authenticity or a confidentiality problem (or both), and then choose the appropriate cryptographic technique for solving the problem.

2.2.2 Unconditional versus Computational security

- Cryptographic instructions may provide unconditional or only computational security

- **Unconditional security** is protection that a computer with unlimited computational power could not break
  - Systems that uses this kind of security is in general not so interesting from a practical point of view
- **Computational security** is that a system could in principle be broken but it would require spending a completely unrealistic amount of computing power

2.3 Distributed Systems

2.3.1 What is a Distributed System?

- A distributed is a collection of computers which collaborate to solve a joint problem
  - They try to appear as one consistent system towards the user
- The book only concentrates on distributed systems with the following properties
  - **Geographic separation:*** The machines in the system are distributed over a large geographic area
    - They are typically located in different administrative domains with different system admins
  - **No common physical clock:** The machines do not share a physical clock
    - They cannot coordinate via some notion of global time
    - They might have local physical clocks which are near impossible to synchronize perfectly
      - Tend to drift apart if nothing is done to synchronize them
  - **Coordination is via message passing:** The machines coordinate via passing messages
    - The communication is assumed to be via TCP/IP
  - **Heterogeneous:** The machines have different hardware, different operating system and different computational power and connectivity.
    - Some might be powerful servers while others are laptops or even cheap sensors

- A distributed system is often provided via a *middleware* layer running on top of the different operating systems at the physically separated machines
- The main goal of the middleware are:
  - **API defined:** The middleware provides some uniform API to the application on all different machines using the distributed system.
    * The behavior of the system is ideally only defined via this API
    * An implementation an be replaced with any other version tat satisfies the API without the application layer noticing
  - **Transparency:** The middleware should hide the quirks of the different operating systems and the fact that the system is distributed.
  - **Openness/Interoperability:** The system should ideally be defined via standards or interfaces such that different operator can implement their own version.

2.3.2 Replication and Consistency

- One of the main challenges of distributed system is **Consistency**
  - When a resource is replicated it should be consistent across all replicas
- Consistency is defined via the input-output behaviour of the distributed system not via the particular internal representation
- The consistency specification is done in two steps
  1. The syntax is given specifying which inputs and outputs the system can receive and give
  2. The required relation between the inputs and outputs is specified

2.3.3 Types of Failure

- **Crash failure:** A machine or process crashed, it stops running now and will never come alive again. There a three main models for this
  - **Fail-arbitrary:** As the process crashes, the last message it sends is some arbitrary possibly faulty message.

  - **Fail-silent:** As the process crashes it sends out no message to any other process.
  - **Fail-stop:** As the process crashes it sends `Stop` to all other processes.

- **Crash-recovery failure:** A machine or process crashed and stops running
  - Later it becomes alive again
  - Can be due to accidentally turn off or burned down and replaced
  - When the machine comes back to life there is typically some reentry protocol that allows it to catch up on what happened.
  - When a machine crashes and then recovers it is assumed that all values in RAM are lost
  - It is typically assumed that values on the disk are uncorrupted.
- **Reset failure:** A machine or process is reset to some previous state.
  - It could for instance forget that a given message was sent a send it again
- **Timing failure:** A machine does not run as fast as expected and sends a given message too late
  - Or runs too fast and sends it too early
- **Message delay failure:** A sent message does not arrive within the time expected
  - The network is too slow
- **Byzantine process failure:** A machine or process is replaced by another process running a possibly maliciously chosen algorithm.
  - This can be due to programming error, hardware error or because the machine is taken over by a hacker.
- **Duplication failure:** A message that should have been sent only once is sent multiple times.
- **Injection error:** A message that should never have been sent was sent, perhaps sent over the network by an adversary trying to break your system.

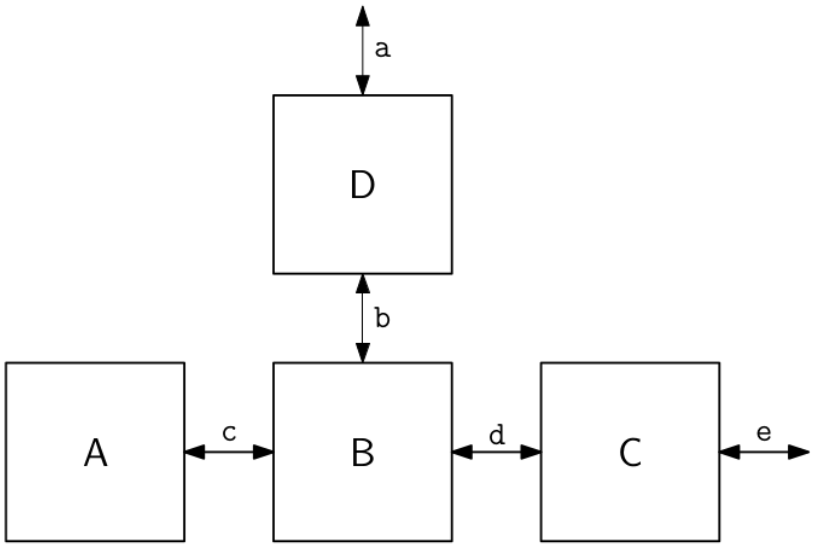
- **Tampering failure:** A message was changed while being sent
  - Possibly by an adversary trying to break your system.
- **Omission failure:** A message that should have been sent was never sent, or it was dropped on the network, or not received at the receiving end.

2.3.4 Correctness, Consistency, Fault tolerance, Security: A rose by any other name would smell as sweet

- **Attack model** denotes the specification of the things that can happen to the system
- To specify what the system is suppose to do we need to specify at least one attack model what the system is suppose to do
- In general we say that under some attack model $\mathcal{A}$ the system behaves according to specification $\mathcal{S}$. Then $(\mathcal{A}, \mathcal{S})$ is the full specification of the system's guarantees. Sometimes a system will have several different behaviour
- The full specification of what we desire from a system is the set $X = \{(\mathcal{A}_i, \mathcal{S}_i)\}$
  - If it holds for a protocol π that for each $(\mathcal{A}, \mathcal{S}) \in X$, π behaves according $\mathcal{S}$ in the attack model $\mathcal{A}$, then we say that π is a secure implementation of X
  - This means that security is always relative to some specification X
- The notions of correctness, consistency and fault tolerance are all just notions of security according to some specification.
  - Correctness is typically just achieving security in an attack model with no or very mild failures.
  - Consistency is just security in the context of databases and replicated system
  - Fault tolerance is security in attack models with bad failures

2.3.5 Specifying and Modelling Distributed Systems

- The systems will be specified via interactive agents (IAs)
- An interactive agent A is a computational device which receives and sends messages on names ports and which hold an internal state



#+CAPTION Figure 1.1 An interactive agent A with a port c; An IA B with ports c and d; An IA C with ports d and e; And an IA D with ports b and a. Together they make up an interactive system with open ports a and e When an IA is run it will produce a trace, which is just the sequence of messa

- When an IA is run it will produce a `trace`
  - Is a sequence of messages exchanged between the agents in order
- Formally start with the empty trace
  - When m is written on port p you append (SEND, p, m) to the trace
  - When m is read on port p you append (READ, p, m) to the trace
- Specification in the book is done via **trace properties** which are just functions P which takes traces as input ad return true or false.

- If τ is a trace, then $P(\tau) = \top$ means that the trace has the property P
- $P(\tau) = \bot$ means that the trace does not have the property P

- There are two important classes of trace properties
  - **Safety property:** It is something which is true until some safety condition is broken at which point it never becomes true again
    - Formally a safety property is a trace property P suchs that $P(\epsilon) = \top$ and if $P(\tau) = \bot$ then $P(\tau \circ \tau') = \bot$ for all traces τ'
  - **Liveness property:** It is something which will eventually become true in some attack model
    - Formally a safety property is a trace property P suchs that $P(\epsilon) = \bot$ and if $P(\tau) = \top$ then $P(\tau \circ \tau') = \top$ for all traces τ'

- All traces properties can be written as a conjunction of safety and liveness properties

2.3.6 Types of Distributed Systems Models

- **Fully synchronous:** Each process P has a clock $c(P)$ and there is a known bound Δ_{CLOCK} on how far they drift from real time.
  - There is also a known upper bound Δ_{SEND} on how long it takes to send a message between any two parties
  - The model often assumes some noting of global time what is used to define what the real time is.
  - N is used to denote nature and $c(N)$ to denote real time: the time of nature
  - Hard to implement - almost impossible
  - Easy to use

- **Fully asynchronous:** The process P do not even have clocks
  - The protocols cannot be specified b asking parties to do things at specific times
  - There are no know upper bound on how long it takes to send a message

  - It is assumed that if a message is sent by a correct process and that process never crashes
  - The delivery is model as follows
    * There is a value Δ_{SEND}
    * If a message is send by a correct process that does not crash then it is delivered within that bound
  - Easy to implement - almost impossible
  - Hard to use

- **Eventually synchronous:** A popular mixed model, where the processors have clocks.
  - Two bounds Δ_{CLOCK} and Δ_{SEND} are given
  - The clocks can be far apart and messages take arbitrarily long to deliver
  - However, it is assumed that eventually there will always come a period where the network is synchronous
  - Medium to implement
  - Medium to use
  - The goals
    * The safety properties are guarantees always
    * The liveness properties are guaranteed when synchronous

2.3.7 Canonical Goals

- **Leader Election:** In many distributed systems, one can gain efficiency by having one server take on a special role, like collecting and distributing tasks.
  - Such a server is often called the leader.
  - The problem is that when the leader crashes, liveness typically breaks down in a very bad way.
    * Then it is time to pick a new leader.
    * This is called leader election.
    * The goal is that all processes end up agreeing on some new non-crashed process that will be the new leader.

- **Broadcast:** You want to send a message m to all machines in the system.
  - This is easy when there are no failures.
  - If there are transmission errors, various systems with acknowledgements and resends will solve the problem.
  - In case of process failures, the problem becomes much harder.
    * It is typically needed that all processes receive the same message, even if the sender and some of the other processes crash or are suffering Byzantine failure.
    * The problem can be made even harder by considering many messages being sent by different parties at the same time and require that they arrive in the same order on all processk
- **Consensus:** All processes have a local input bit 0 or 1.
  - This could for instance be a bit indicating whether they are all ready to proceed in the protocol.
  - They should end up outputting a common decision, i.e., there is a bit b such that all correct processes eventually output b.
  - There are several variations on how this output depends on the inputs.
- **MUTEX:** Mutual exclusion. The processes run the same code.
  - There is some critical region in the code that only one server is allowed to enter at a time.
  - This problem is already hard in multi-threaded programs.
  - In becomes much worse in a distributed system where the processes are physically separated.

3 Communication (2)

3.1 Types of communication

- **Message Passing:** Processes can send messages to each other
  - The transfer is typically between two parjjjjjjjties only (the sender and receiver)
  - Sending and receiving are separated in time

- **Shared Memory Space:** Processes are in a setting where they can see the same variables
  - They communicate by writing and reading these variables
  - Different cores and threads on the same computer communicate via this

3.2 Lets Go!

```
package main

import ( "fmt" )

// A function declaration: Types after variable names; Multiple return values
func multi(x int, y int) (string, int) {
return "The answer is", x+y
}

// A class
type Named struct {
name string // Member of a class
}

// Method on a class
func (nm *Named) PrintName(n int) {
if n < 0 { panic(-1) } // an "exception"
for i:=0; i<n; i++ {
fmt.Print(nm.name + "\n")
}
}

func main() {
var i int // Explicit declaration of variable
i = 21
// = is used for assignment
j := 21
// := declares the variable based on the type of the value
decr := func() int { // a closure
j = j-7
return j
```

```
}
str, m := multi(i, j)
defer fmt.Println(str, m) // run after return or a panic
fmt.Println(decr())
fmt.Println(decr())
nm1 := Named{ name: "Jesper" } // value
nm2 := &Named{} // pointer
nm1.PrintName(2) // Calling a method
nm2.PrintName(2) // . does auto-dereference of pointer
nm2.name = "Claudio" ; nm2.PrintName(2) // RETURN or ; as seper
}
```

3.3 The Internet

3.3.1 IP

- A Go program which will display the name and IP address of your machine

```
package main

import (
"fmt"
"net"
"os"
"strconv"
)

func main() {
name, _ := os.Hostname()
addrs, _ := net.LookupHost(name)
fmt.Println("Name: " + name)
for indx, addr := range addrs {
fmt.Println("Address number " + strconv.Itoa(indx) + ": " + addr)
}
}
```

3.3.2 UDP

- An UDP Server which receives UDP packages and prints then

```
package main

import (
"fmt"
"net"
)

func main() {
ServerAddr, _ := net.ResolveUDPAddr("udp", ":10001")
ServerConn, _ := net.ListenUDP("udp", ServerAddr)
defer ServerConn.Close()
buf := make([]byte, 1024)
for {
n, addr, err := ServerConn.ReadFromUDP(buf)
fmt.Println("Received ", string(buf[0:n]), " from ", addr)
if err != nil {
fmt.Println("Error: ", err)
}
}
}
```

- A UDP client sending packages to port 10001 on the local machine

```
package main

import (
"net"
"strconv"
"time"
)

func main() {
ServerAddr, _ := net.ResolveUDPAddr("udp", "127.0.0.1:10001")
LocalAddr, _ := net.ResolveUDPAddr("udp", "127.0.0.1:0")
conn, _ := net.DialUDP("udp", LocalAddr, ServerAddr)
defer conn.Close()
i := 0
for {
i++
msg := strconv.Itoa(i)
```

```
conn.Write([]byte(msg))
time.Sleep(time.Second * 1)
}
}
```

3.3.3 TCP

- A TCP Server

```
package main

import ( "net" ; "fmt" ; "bufio" ; "strings" )

func handleConnection(conn net.Conn) {
defer conn.Close()
for {
msg, err := bufio.NewReader(conn).ReadString('\n')
if (err != nil) {
fmt.Println("Error: " + err.Error())
return
} else {
fmt.Print("From Client:", string(msg))
titlemsg := strings.Title(msg)
conn.Write([]byte(titlemsg))
}
}
}

func main() {
fmt.Println("Listening for connection...")
ln, _ := net.Listen("tcp", ":18081")
defer ln.Close() // Makes the function ln.Close() run after main terminates
conn, _ := ln.Accept()
fmt.Println("Got a connection...")
handleConnection(conn)
}
```

- A TCP Client

```
package main
```

```
import (
"bufio"
"fmt"
"net"
"os"
)

var conn net.Conn

func main() {
conn, _ = net.Dial("tcp", "127.0.0.1:18081")
defer conn.Close()
for {
reader := bufio.NewReader(os.Stdin)
fmt.Print("> ")
text, err := reader.ReadString('\n')
if text == "quit\n" {
return
}
fmt.Fprintf(conn, text)
msg, err := bufio.NewReader(conn).ReadString('\n')
if err != nil {
return
}
fmt.Print("From server: " + msg)
}
}
```

- A go program which starts listening on a random port

```
package main

import (
"fmt"
"net"
)

func main() {
ln, _ := net.Listen("tcp", ":")
defer ln.Close()
```

```
_, port, _ := net.SplitHostPort(ln.Addr().String())
fmt.Println("Listening on port " + port)
}
```

3.3.4 DNS

- Look up the IP address of `https://www.google.com`

```
package main

import (
"fmt"
"net"
"strconv"
)

func main() {
addrs, _ := net.LookupHost("google.com")
for indx, addr := range addrs {
fmt.Println("Address number " + strconv.Itoa(indx) + ": " + addr)
}
}
```

- Ask Google a question

```
package main

import (
"bufio"
"fmt"
"net"
)

func main() {
addrs, _ := net.LookupHost("www.google.com")
addr := addrs[0]
fmt.Println(addr)
conn, err := net.Dial("tcp", addr+":80")
if conn != nil {
defer conn.Close()
}
```

```
if err != nil {
panic(0)
}
fmt.Fprintf(conn, "GET /search?q=Secure+Distributed+Systems HTTP/1.1\n")
fmt.Fprintf(conn, "HOST: www.google.com\n")
fmt.Fprintf(conn, "\n")
for {
msg, err := bufio.NewReader(conn).ReadString('\n')
if err != nil {
panic(1)
}
fmt.Println(msg)
}
}
```

3.4 Concurrency

- To coordinate concurrent threads in Go, two abstractions is used MUTEX and channels
  - MUTEX has the type `Mutex` and has two methods `Mutex.Lock()` and `Mutex.Unlock()`
    - Can be held by a single writer or multiple readers
  - A channel has a type which is the type of objects that can be send over the channel
    - Any goroutine can send on a channel and any goroutine can receive from a channel.
    - It is safe for many goroutines to send and receive at the same time without using MUTEXs.
    - When a thread sends on a channel it blocks until the message is read
    - The build in map type is not thread safe
- A multithreaded TCP server

```
package main

import (
"bufio"
"fmt"
```

```
"net"
"strings"
)

func handleConnection(conn net.Conn) {
defer conn.Close()
myEnd := conn.LocalAddr().String()
otherEnd := conn.RemoteAddr().String()
for {
msg, err := bufio.NewReader(conn).ReadString('\n')
if err != nil {
fmt.Println("Ending session with " + otherEnd)
return
} else {
fmt.Print("From " + otherEnd + " to " + myEnd + ": " + string(msg))
titlemsg := strings.Title(msg)
conn.Write([]byte(titlemsg))
}
}
}

func main() {
ln, _ := net.Listen("tcp", ":18081")
defer ln.Close()
for {
fmt.Println("Listening for connection...")
conn, _ := ln.Accept()
fmt.Println("Got a connection...")
go handleConnection(conn)
}
}
```

- A use of channels in Go

```
package main

import (
"fmt"
"strconv"
)
```

```
var sendernames = [5]string{"Alice", "Bob", "Chloe", "Dennis", "Elisa"}
var receivernames = [5]string{"Frederik", "Gary", "Hailey", "Isabel", "Jesper"}

func send(c chan string, myname string) {
for i := 0; i < 1000; i++ {
// you send on a channel using <-
c <- myname + "#" + strconv.Itoa(i)
}
}

func receive(c chan string, myname string) {
i := 0
for {
// you also receive from channel using <-
msg := <-c
fmt.Println(myname + "#" + strconv.Itoa(i) + " " + msg)
i++
}
}

func main() {
c := make(chan string)
for i := 0; i < 5; i++ {
go send(c, sendernames[i])
go receive(c, receivernames[i])
}
// we do one blocking call to avoid that the program terminates
receive(c, "Kacey")
}
```

- A Go program with a race condition

```
package main

import (
"fmt"
"sync"
)
```

```
type DNS struct {
m    map[string]string
lock sync.RWMutex
}

func (dns *DNS) Set(key string, val string) {
dns.lock.Lock()
defer dns.lock.Unlock()
dns.m[key] = val
}

func (dns *DNS) Get(key string) string {
dns.lock.RLock()
defer dns.lock.RUnlock()
return dns.m[key]
}

func MakeDNS() *DNS {
dns := new(DNS)
dns.m = make(map[string]string)
return dns
}

func GetAndSet(suf string) {
for i := 0; i < 10; i++ {
dns.Set("X", dns.Get("X")+suf)
}
c <- 0
}

var c = make(chan int)
var dns = MakeDNS()

func main() {
go GetAndSet("1")
go GetAndSet("2")
go GetAndSet("3")
<-c
<-c
<-c // wait for the three goroutine
```

```
}
```

- A Go program without a race condition (using locks)

```
package main

import (
"fmt"
"sync"
)

type DNS struct {
m    map[string]string
lock sync.Mutex
}

func MakeDNS() *DNS {
dns := new(DNS)
dns.m = make(map[string]string)
return dns
}

func (dns *DNS) GetAndSetOnce(suf string) {
dns.lock.Lock()
defer dns.lock.Unlock()
dns.m["X"] = dns.m["X"] + suf
}

func (dns *DNS) GetAndSet(suf string) {
for i := 0; i < 10; i++ {
dns.GetAndSetOnce(suf)
}
c <- 0
}

var c = make(chan int)

func main() {
dns := MakeDNS()
go dns.GetAndSet("1")
```

```
go dns.GetAndSet("2")
go dns.GetAndSet("3")
<-c
<-c
<-c // wait for the three goroutines to end
fmt.Println(dns.m["X"])
}
```

3.5 Marshalling

- Go has a serializability system called `Gob`
  - In go there is a notion that a field of a struct being exported or not
    - If the name of the field starts with a capital letter it is exported
  - Only exported fields are send in Gob
- A TCP server using Gob

```
package main

import (
"encoding/gob"
"fmt"
"io"
"log"
"net"
)

type ToSend struct {
Msg    string // only exported variables are sent, so start the ...
Number int    // ... name of the fields you want send by a capital letter
}

func handleConnection(conn net.Conn) {
defer conn.Close()
msg := &ToSend{}
dec := gob.NewDecoder(conn)
for {
```

```
err := dec.Decode(msg)
if err == io.EOF {
fmt.Println("Connection closed by " + conn.RemoteAddr().String())
return
}
if err != nil {
log.Println(err.Error())
return
}
fmt.Println("From "+conn.RemoteAddr().String()+":\n", msg)
}
}
func main() {
fmt.Println("Listening for connection...")
ln, _ := net.Listen("tcp", ":18081")
defer ln.Close()
conn, _ := ln.Accept()
fmt.Println("Got a connection from ", conn.RemoteAddr().String())
handleConnection(conn)
}
```

- A TCP client using Gob

```
package main

import (
"bufio"
"encoding/gob"
"fmt"
"net"
"os"
)

type ToSend struct {
Msg    string // only exported variables are sent, so start the ...
Number int    // ... name of the fields you want sent by a capital letter
}

func main() {
ts := &ToSend{}
```

```
conn, _ := net.Dial("tcp", "127.0.0.1:18081")
defer conn.Close()
enc := gob.NewEncoder(conn)
for i := 0; ; i++ {
fmt.Print("> ")
reader := bufio.NewReader(os.Stdin)
m, err := reader.ReadString('\n')
if err != nil || m == "quit\n" {
return
}
ts.Msg = m
ts.Number = i
enc.Encode(ts)
}
}
```

3.6 Remote Procedure Calls

- Also sometimes called RMI in object oriented languages
- Remote Procedure Calls works as follows:
    1. The parameters are sent over the network to a remote machine
    2. The function is run on some remote value is returned
    3. The return values are sent over the network to the caller using serialization
- An RPC server

```
package main

import (
"bufio"
"fmt"
"log"
"net"
"net/rpc"
"os"
"time"
)
```

```
type PrintAndCount struct {
x int
}

func (l *PrintAndCount) GetLine(line []byte, cnt *int) error {
HardTask()
l.x++
fmt.Println(string(line))
*cnt = l.x
return nil
}
func HardTask() {
time.Sleep(5 * time.Second)
}
func MakeTCPListener() *net.TCPListener {
addy, err := net.ResolveTCPAddr("tcp", "0.0.0.0:42587")
if err != nil {
log.Fatal(err)
}
inbound, err := net.ListenTCP("tcp", addy)
if err != nil {
log.Fatal(err)
}
return inbound
}
func main() {
// Register how we receive incoming connections
go rpc.Accept(MakeTCPListener())
// Register a PrintAndCount object
rpc.Register(new(PrintAndCount))
// Avoid terminating
fmt.Println("Press any key to terminate server")
bufio.NewReader(os.Stdin).ReadLine()
}
```

- An synchronous RPC client

```
package main

import (
```

```
"bufio"
"fmt"
"log"
"net/rpc"
"os"
)

func main() {
client, err := rpc.Dial("tcp", "localhost:42587")
if err != nil {
log.Fatal(err)
}
in := bufio.NewReader(os.Stdin)
for {
line, _, err := in.ReadLine()
var reply int
// Synchronous call
err = client.Call("PrintAndCount.GetLine", line, &reply)
if err != nil {
log.Fatal(err)
}
fmt.Println("Strings printed at server: ", reply)
}
}
```

- An asynchronous RPC client

```
package main

import (
"bufio"
"fmt"
"log"
"net/rpc"
"os"
)

func PrintWhenReady(call *rpc.Call) {
<-call.Done // this channel unblocks when the call returns
if call.Error != nil {
```

```
log.Fatal(call.Error)
}
fmt.Println("Strings printed at server: ", reply)
}

var reply int

func main() {
client, err := rpc.Dial("tcp", "localhost:42587")
if err != nil {
log.Fatal(err)
}
in := bufio.NewReader(os.Stdin)
for {
line, _, _ := in.ReadLine()
// Asynchronous call
call := client.Go("PrintAndCount.GetLine", line, &reply, nil)
go PrintWhenReady(call) // Handles the reply when ready
fmt.Println("See, I can still do stuff!")
fmt.Println("See, I can still do stuff!")
fmt.Println("OK, I'm bored...")
}
}
```

3.7 Logically Organising Connections

- **Fully Connected:** Let everyone in the system make a connection to every else
  - Often done in smaller systems where a server is replicated
  - Does not scale well because with n servers the number of connections is in the order n^2
- **Client-Server:** There is a single server to which all the clients connect
  - There is a single point of failure
    - If the server crashed, is attacked or is corrupted the security of the system breaks down
- **Client-Replicated Server:** Replicated servers
  - Deals with the single point of attack problem

- Most common architecture on the internet
- Opens up a lot of problems with keeping the copies of the replicated server consistent

- **Edge nodes:** All access is meditated by edge nodes
  - Client contact the nodes which distribute the work load over the replicates
  - Useful of replicated servers

3.8 Unstructured Peer-to-Peer

3.8.1 General

- There are two main kinds of peer-to-peer networks structured and unstructured
  - In structured the connection between nodes are established in a deterministic way
  - In unstructured the connections are formed at random

- A peer-to-peer graph can be used to disseminate messages.
  - If you want to send a message, simply send it to the peers you have chosen.
  - If a node received a message it did not see before then it recursively disseminate it the same way.

- There are two measure of a how good a peer-to-peer graph are
  - **Diameter:** For each pair (P_i, P_j) of parties, define their distance to be the shortest path from P_i to P_j. The diameter of the graph is

$$\text{Diameter} = \max_{P_i \ P_j} \text{Distance}(P_i, P_j) \tag{1}$$

- If they are not connected the distance is $+\infty$
- To compute the distance from two parties one can flood the network starting from P_i to see how long it takes to reach P_j

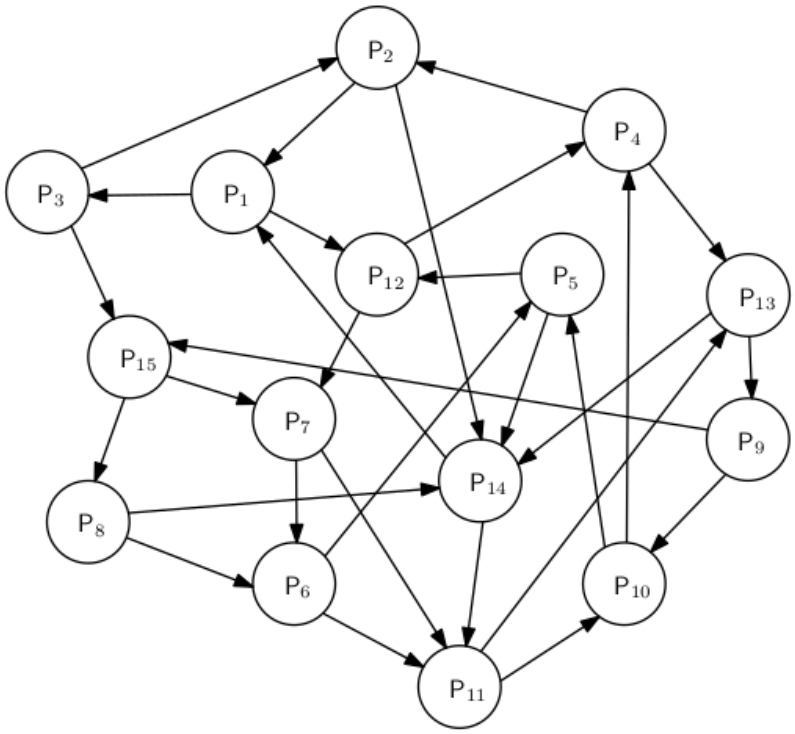


Figure 1: Random Peer-to-Peer

- **Connectivity:** It is the minimal number of edges that must be removed to make the graph unconnected

- The reason one might use a random peer-to-peer network is that they are easy to form

- **THEOREM 2.1** Assume that $n \geq 25$ and $\kappa \geq 2$. If you pick a random peer-to-peer graph on n parties with out degree κ, then the probabilty that it is not connected is at most $2^{-\kappa}$

- An **Eclipsing Attack** is when the adversary controls some nodes $A \subset P$ and removes the connection and creates a cut

  - A type of Byzantine Attack

3.8.2 How to Build and Maintain

- To build a random peer-to-peer network, each node could hold a list of all other nodes present in the network and pick its connections random from this list

  - When a new node enters the network it will flood its presence and all other nodes will add it to their network list
  - If a node becomes unresponsive, nodes will remove it from their network list.
  - To enter the network initially, a node needs to know at least one other node of the network and ask it for its network list.

- In practice it is not possible to have a list of every node on the network, so there are other methods to build a random look network.

  - Each node only holds a partial view.
  - When entering the network you get the partial view of your entry point.
  - Then you can contact some of the know peers to get their partial view and that way learn about more peers, and yourself build a random looking partial view of the system.

3.9 Structured Peer-To-Peer

- A popular structured peer-to-peer network is the Chord network
  - Each node has an identity
    - e.g. IP-address
  - The identity is mapped deterministically to an integer between 0 and $B - 1$ for some bound B
    - B has to large enough so that the change that a node is mapped to the same integer is small
  - point as 0, B + 1 would hit the same point as 1 and so on.
  - When a number of nodes with identities $A, B, C \ldots$ enter the network, they will each map their identity into a number.
  - Each node is ordered in a circle
  - Each node will then make a connection the node that is 1 step away on the circle, 2 steps away, 4 steps away8 steps away, 16, 32, and so on.
  - If there is no node at those exact points they make a connection to the next node on the circle.
  - This ensures that the diameter is only $\log_2(B)$ so that it does not take to long to sent a message

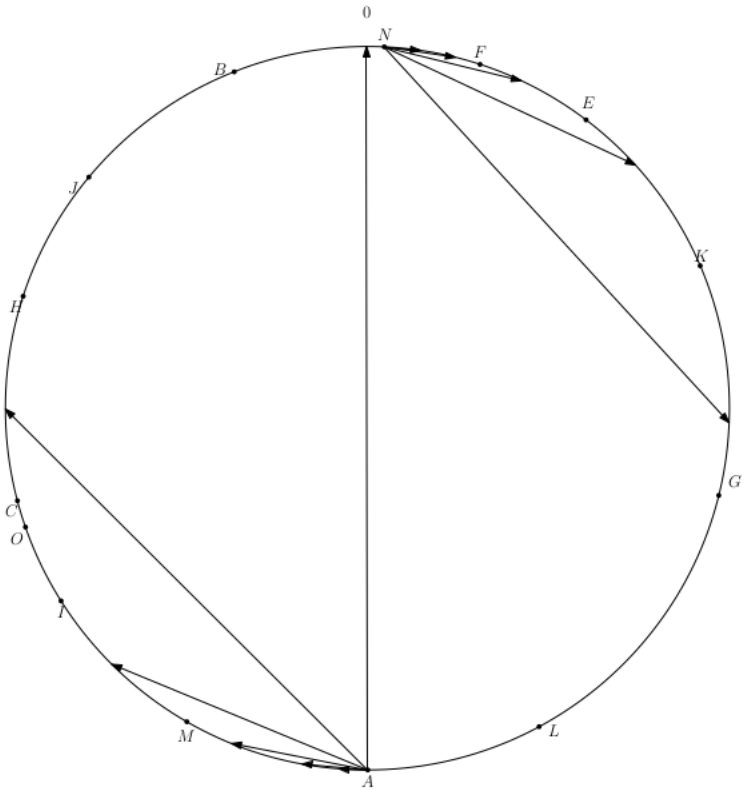


Figure 2.26 Chord Illustration

4 A Syntax for Distributed Systems (3)

4.1 Basic System Model

4.1.1 Interactive Agents

- The most basic entity will be an Interactive Agent (IA), which is formally defined by a tuple $(\mathcal{P}, \sigma_0, T)$ where:
  - $\mathcal{P}$ is a finite set containing the ports which the IA can receive inputs and outputs
    - Each ports can be used for both sending and receiving
  - σ_0 is the initial state of the IA
  - T is the transition function

        * It says what the system does in response to a message being received on one of the ports
        * Depends on the current state σ, the port name $P \in \mathcal{P}$ and the message m being received
        * The output specifies the new current state σ', a new port $Q \in \mathcal{P}$ and some new message $m' \in \mathcal{P}$
            · It is written $(\sigma', Q, m') \leftarrow T(\sigma, P, m)$
            · The transition function always send a new message on some legal port

- The IA A is thought of as a box holding its current state σ
- The syntax $(Q, m') \leftarrow A(P, m)$ is used to mean: fetch the current state σ from A and compute $(\sigma', Q, m') \leftarrow T(\sigma, P, m)$ and replace the current state in A by σ'
- The transition function is allowed to be randomised
    - This allows to model an algorithm which for instance samples a random RSA key (n, e, d), saves (n, d) in the state σ' and sends (n, e) on some port.
- The transition function should not depend on port names
    - Done to avoid conflicting port names

4.1.2 Interactive Systems

- An interactive system (IS) S is a set of IAs
    - If there are two IAs with the same port name, they are thought of as being connected
    - If there are more than two IAs with the same port name the system is **malformed** and $S = \perp$ is written
    - Composing two systems S_1 and S_2 is done by taking the set union $S_1 + S_2 = S_1 \cup S_2$
- The ports not connected to other ports are called **open ports**
    - An IS can be activated on an open port P by sending some message m
- If (P, m) is inputted to S, the execution proceeds as follows

1. Let A_0 be the IA with port P, let $m_0 = m$ and $P_0 = O$, e then compute $(P_1, m_1) \leftarrow A_0(P_0, m_0)$
2. If P_1 is a closed port, then let A_1 be the other IA with port P_1. We then compute $(P_2, m_2) \leftarrow A_0(P_1, m_1)$ and so on until we compute $(P_{t+1}, m_{t+1}) \leftarrow A_t(P_t, m_t)$ and P_{t+1} is an open port
3. Then let $Q = P + 1$ and $m' = m_{t+1}$ return (Q, m')

- When compute an output the IS also updates its internal state of the system
  - An IS is therefore in some sense just a more structured IA

4.2 Modelling Persistent Storage

Ports The are two ports write and read.
Init When the disk is initialised it creates a map $M : A \to B$ between sets A and B. We let $\perp$ denote that an entry is uninitialised and we assume that $\perp \in B$. Initially set $M[x] = \perp$ for all $x \in A$.
Write When it receives a value $(x, v) \in A \times B$ on write it sets $M[x] = v$ and returns DONE on write.
Read When it receives a value $x \in A$ on read it returns $M[x]$ on read.

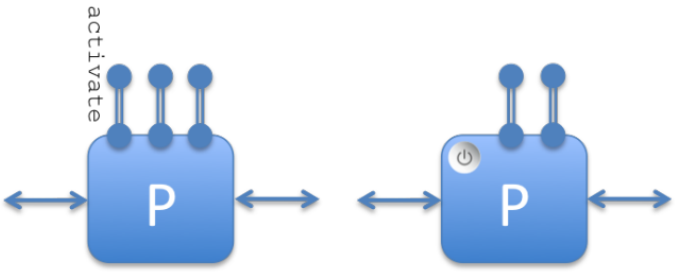
Figure 2: A Disk. To the left, the structure of the disk. To the right, how we will depict it in the rest of the text.

- The book gives an oversimplified model of persistent storage
  - Implementing it in practise is a whole science in itself
  - Atomicity is for example hard to implement

4.3 Modelling Processes

4.3.1 General

- The model of processes given is based solely on message passing and pure asynchronous programming



Protocol Ports versus Special Ports The ports of the process are divided into the protocol ports and the special ports. Protocol ports correspond to the ports the process has in "real life". The special ports are used only for modeling purpose. They are for instance used to model when a process is scheduled, when it crashes, and so on. We draw special ports with rounded ends.

Handling Input For each protocol port P the process has a queue $P.Q$. Initially the queue is empty. When the process receives a message m on a protocol port P it adds the message to $P.Q$. Then it returns DONE on P.

Activate The process has a special port called activate. When it receives any value on activate it can do one of the following any number of times:

- Read from a message queue $P.Q$.
- Change its internal state in any other way.
- Do a recursive call on some other port Q.

The activation ends when it returns any value on activate.

Figure 3: A Process. To the left, its structure. To the right, how we depict it in the rest of the text.

  - All inputs to a process are given as messages and are queued on the receiving process in such a way that the "calling" process can proceed its execution immediately.
  - The caller never waits on its input

- A process will receive its input asynchronously as follows
  - When a message m is received on some port P, the message m is added to queue Q, then process returns and does nothing more
  - There is a separate "thread" associated to the process which does the actual work
    * It proceeds in steps known as activations
    * In each activation it is allowed to do some small amount of work and then the activation stops again

- To model that processes by default might not progress at the same speed, a special port is introduced on which the process is told when to do the activations.
  - Think of some evil daemon being connected to these ports.
  - We cannot by default assume that all processes proceed at the same speed, so to be on the safe side we assume that the order in which they take steps could be the worst possible one.

4.3.2 Specifying a Process

- Formally a process is given by a transition function
  - A very cumbersome formalism

- The process is instead specified by a list of activation rules written in prose or pseudo-code, some examples of activation rules:
  - On input $(\text{INCREMENT}, a)$ on I, let $x \leftarrow x + a$ and send x on port P
  - On input (PING) on P return on P
  - On input $(\text{CONDITIONAL-INCREMENT}, a, b,)$, if $x \leq b$, let $x \leftarrow x + a$ and send x on port P

- In its more general form an activation rule has the format /"On input $(\text{NAME}, v_1, v_2, \ldots,)$ on port P, if Cond, do A"

- The rule is `triggerable` if in the queue $P.Q$ there is a message m of the form $(\text{NAME}, v_1, v_2, \ldots,)$ and at the same time the condition Cond is true
- The condition might depend on the message and the internal state of the process
- If the rule is triggered the m is removed from the queue, if not the message is not removed.
- If the rule is triggered then the algorithm A is run
  - This algorithm is allowed to run as a normal activation of a process
  - After running A the process return on `activate`
- When a process is activated port `activate` it goes though all its activation rules from top to bottom
  - If it finds on which is triggerable it triggers that rule
  - If no activation rule is triggerable it does nothing and returns immediately on `activate`
  - If no activation rule is triggerable we say the process is `idle`

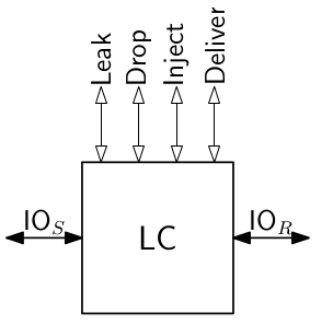
4.3.3 Some Possible Special Ports

- In addition to `activate` a process might have other special ports.
  - These special ports are typically used to model faults.
- Examples of special ports.
  - **crash:** On any input on the crash port, delete the process' set of activation rules.
    - This means that the machine no longer does anything when activated.
  - **takeover:** On input `AR` on takeover the activation rules of the process are replaced by the activation rules in AR.
    - If the process models a machine, then this corresponds to a hacker completely taking over the machine and installing its own code on the machine.
  - **leak:** On input L on leak, let σ denote the internal state of the process, including all the queues of incoming messages. Compute $y = L(\sigma)$. Then return y on leak.

* This corresponds to the process leaking some information to its environment.
* If the leakage function L is the identity, then it corresponds to a hacker breaking into the machine and seeing all the data on the machine.
* In some case more limited leakage makes sense too.

- By default a process only has the `activate` special port

4.4 Modelling Cnhannels



Ports The channel connects two parties with names S and R. For each party $P \in \{S, R\}$ it has a port called IO_P. It has a special port called Leak, which is used to model that the channel does not hide what is sent on it from its environment. It has a special port Drop, which is used to model that it can drop messages. It has a special port INJECT, which is used to model that anyone can inject messages on the channel: it is not authenticated. Finally, it has a special port Deliver which is used to model that it is its environment which controls when messages are delivered.

Init It keeps a set InTransit which is initially empty.

Send On input (R, m) on IO_S, it output (S, R, m) on Leak, adds (S, R, m) to InTransit and then it returns on IO_S. Similarly for IO_R.

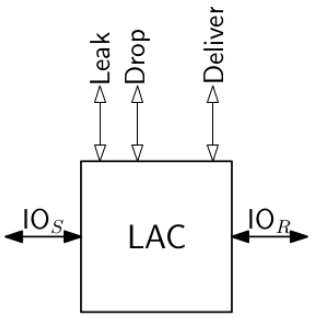
Drop On input (P, Q, m) on Drop it removes (P, Q, m) from InTransit and returns on Drop.

Inject On input (P, Q, m) on INJECT it adds (P, Q, m) to InTransit and returns on INJECT.

Deliver On input (P, Q, m) on Deliver where $(P, Q, m) \in$ InTransit, it removes (P, Q, m) from InTransit, outputs (P, m) on IO_Q and returns on Deliver.

Figure 4: Lossy Channel

- The same distinction between protocol ports and special ports is used



Ports The channel connects two parties with names S and R. For each party P it has a port called IO_P. It has a special ports Leak, Drop, and Deliver.

Init It keeps a set InTransit which is initially empty.

Send On input (R, m) on IO_S, it output (S, R, m) on Leak, adds (S, R, m) to InTransit and then it returns on IO_S. Similarly for IO_R.

Drop On input (P, Q, m) on Drop it removes (P, Q, m) from InTransit and returns on Drop.

Deliver On input (P, Q, m) on Deliver where $(P, Q, m) \in$ InTransit, it removes (P, Q, m) from InTransit, outputs (P, m) on IO_Q and returns on Deliver.

Figure 5: Lossy Authenticated Channel

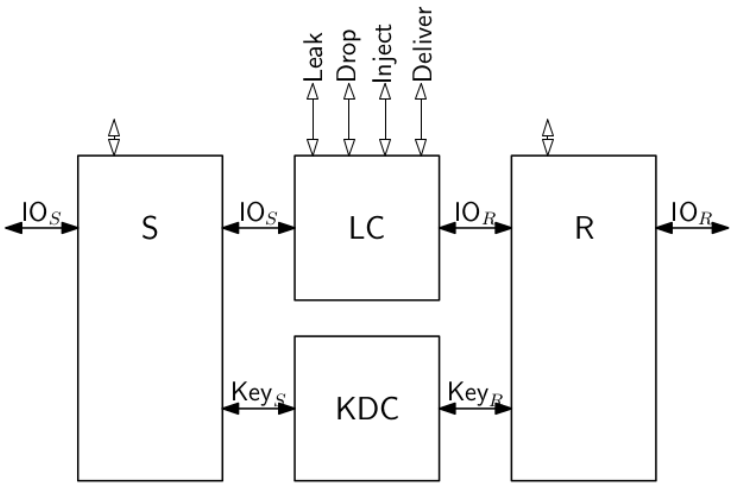
Ports The channel connects two parties with names S and R. For each party $P \in \{S, R\}$ it has a special port Key_P.

Generate Key It is parametrised by a key generation algorithm Gen. When activated on Deliver the first time it samples a key $K \leftarrow$ Gen.

Deliver On input on Key_S output K on Key_S. On input on Key_R output K on Key_R.

Figure 6: KDC

4.5 Modelling Protocols



Both parties $P \in \{R, S\}$ run this code:

- If no K is stored, call KDC.Key if this was not already done.
- On output K on KDC.Key, store K.
- On (Q, m) on LC2LAC.IO$_P$, add (Q, m) to InTransit.
- If K is stored and there is some $(Q, m) \in$ InTransit then compute $c = \mathrm{MAC}_K(P, Q, m)$ and input (m, c) on LC.IO$_P$.
- On output $(Q, (m, c))$ on LC.IO$_P$ where K is stored check that $c = \mathrm{MAC}_K(Q, P, m)$. If so, output (Q, m) on LC2LAC.IO$_P$.

Figure 3.9 The LC2LAC protocol. It uses two channels, LC and KDC. It has two processes, S and R. They are connected to the channels as shown in the figure. Party P in addition has a port called LC2LAC.IO$_P$ (in the figure just called IO$_P$. The special ports on the parties are there simply as placeholders to remind us that the parties too could have special ports to for instance model corruption.)

- Protocols wil consist of processes, channels and disks connected to to each other in some way

4.6 Specifying Safety Properties

- Safety properties asically specify what is the correct relation between the inputs and the outputs of the system
  - A safety property is only broken if the system gives a wrong output

- **Definition 3.1 (Security):** We say that a protocols π securely implements a channel C if there exists an interactive agent Sim that connects only to the special ports of C and such that π and $C + \text{Sim}$ have the same input output behavior. We write $\pi \sqsubseteq C$

4.7 Specifying Liveness Properties

- Liveness properties specify that the system must make progress
  - e.g. that there are no deadlocks
  - A liveness property is only broken if the system ends up in a state where there is still work to do but the state of the system prevents it from making progress again

4.8 Composition

- Composability is an essential property of any definition of security.
  - If composability holds in the model, then we can prove the properties of the combined protocol in a modular way.
  - This allows us to given a simpler and modular analysis where each step focuses only on one small part of the system (for instance, turning a lossy channel into a non-lossy channel).

5 Consistent Communication (4)

5.1 General

- In many types of networks message can overtake each other
  - This means that you might receive a message A which depends on another message B before you receive V
  - It might also be you don't receive messages from P_1 in the order which they where send by P_1
- It is assumed that there is access to a flooding system
  - It guarantees that if a correct party sends a message then it is eventually delivered at all correct processes
- A system with the following syntax is considered, indicating how a distributed system interacts with the rest of the worlds

- **Send:** A party P_i can get input on the for (P_i, m).
- We say that P_i sent m
- A correct process will physically send a message as soon and it gets it as input
- **Deliver** A party P_i can give output of the form (P_j, m) for $P_j \neq P_i$.
- We say that the message m sent by P_j was delivered by P_i, which is *not* the same event as when the message physically arrives at P_i

- It is assumed that all messages sent by a given party P_i are distinct such that (P_i, n) uniquely identifies a *"sent event"*
- It is assumed that all messages are immediatly delivered local
- It is assumed that the network start from the following property
  - **Liveness** If a correct P_i send (P_i, m), then eventually all correct P_j deliver (P_i, m)

5.2 First In, First Out

- **FIFO:** If a correct P_i sends (P_i, m) and later sends (P_i, m'), then it holds for all correct P_j that if they deliver (P_i, m') then they delivered (P_i, m) earlier
- The following protocol is considered
  1. Initially each party P_i will initialize a counter $c_i = 0$.
     - This counter keeps track of how many messages the party P_i has already sent.
     - Each party P_i also initializes n counters $r_{i,j} = 0$ for $j = 0, \ldots, n$
       - These counters keep track of how many messages party P_i has received from party P_j.
  2. P_i: When sending a message x, send (P_i, c_i, x) on the flooding network and let $c_i = c_{i+1}$.
     - This mean that we tag each message with how many messages were sent before it.

3. P_i: When receiving (P_j, c_j, x) store it in a buffer until $r_{i,j} = c_j$. Then let

$r_{i,j} = r_{i,j+1}$ and output (P_j, m).

- If all parties are correct the FIFO protocol ensures that all messages from each P_i are delivered by all other parties in the order they where send by P_i

5.3 Causality

5.3.1 The Causal-Past Relation

- The Causal-Past Relation is a binary relation $\hookrightarrow$ on messages (P, m)
  - It is written as $(P_i, m_i) \hookrightarrow (P_j, m_j)$
  - It means that in a particular run on the system m_j might depend on m_i
  - e.g. because P_j received m_i before it sent m_j
  - Note that the fact that P_j received m_i before sending m_j does not necessarily mean that m_j in fact depends on m_i , only that it might depend on m_i.
  - Not all sent events are a Causal-Past Relation
- CausalPast(P_j, m_j) is used to denote the set of (P_i, m_i) on which (P_j, m_j) may depend i.e.

$$\text{CasualPast}(P_j, m_j) = \{(P_i, m_i) | (P_i, m_i) \hookrightarrow (P_j, m_j)\} \tag{2}$$

- Method that keeps track of auxiliary sets CausalPast(P_i)
  1. Initially let CausalPast$(P_i) = \emptyset$ for all P_i
  2. On input (P_i, m) at P_i, let CausalPast(P_i) = CausalPast$(P_i) \cup \{(P_i, m)\}$ and let CausalPast(P_i, m) = CausalPast(P_i)
  3. On output (P_j, m) at P_i, let CausalPast(P_i) = CausalPast$(P_i) \cup$ CausalPast(P_j, m)
- $\hookrightarrow$ has the following properties
  - **Transitive:** If $(P_i, m_i) \hookrightarrow (P_j, m_j)$ and $(P_j, m_j) \hookrightarrow (P_k, m_k)$ then $(P_i, m_i) \hookrightarrow (P_k, m_k)$
  - **Reflective:** For all messages it holds that $(P_i, m_i) \hookrightarrow (P_i, m_i)$
  - **Antisymmetric:** If $(P_i, m_i) \hookrightarrow (P_j, m_j)$ and $(P_j, m_j) \hookrightarrow (P_i, m_i)$ then $(P_i, m_i) = (P_j, m_j)$

5.3.2 A Protocol for Causally Consistent Communication

- The safety property of causal order flooding
  - **Causal Consistency:** If $(P_i, m) \hookrightarrow (P_j, m')$ then it holds for all correct P_k that if they deliver (P_j, m'), then they have previously delivered (P_i, m)
- A very inefficient but trivially safe protocol with has the safety property of causal order flooding
  1. Initially let CausalPast$(P_i) = \emptyset$ for all P_i and let Delivered$(P_i) = \emptyset$
  2. On input (P_i, m) at P_i, let CausalPast(P_i) = CausalPast$(P_i) \cup \{(P_i, m)\}$ and let CausalPast(P_i, m) = CausalPast(P_i). Then send (P_i, m) and send along CausalPast(P_i, m)
  3. On receiving (P_j, m) at P_i along with CausalPast(P_j, m) wait until CausalPast$(P_j, m) \subseteq$ Delivered$(Pi) \cup \{(P_j, m)\}$. Then deliver (P_j, m) and let CausalPast(P_i) = CausalPast$(P_i) \cup$CausalPast(P_j, m). Also add (P_j, m) to Delivered(P_i)

5.3.3 Vector Clocks

- Vector clocks improve trivial safety protocol
- Instead of maintaining his causal past each P_i maintains an array VectorClock(P_i) of integers
  - VectorClock$(P_i)[P_k]$ is the number of messages that have currently been received from P_k
  - When a message m_i, P_i will append the current state of his vector clock to the message
    - Denoted VectorClock(P_i, m_i)
  - VectorClock$(P_i, m_i)[P_k]$ is the number of messages P_i has received from P_k at the time he sent m_i
- Instead of remembering all message that were delivered each P_i will maintain an array Delivered(P_i) of the same time as vector clocks
  - Where Delivered$(P_i)[P_k]$ contains the number of messages from P_k that were delivered

- For two vector clocks VectorClock(P_i, m_i) and VectorClock(P_j, m_j) we define

$$\text{VectorClock}(P_i, m_i) \leq \text{VectorClock}(P_j, m_j) \tag{3}$$

to mean that

$$\forall P_k(\text{VectorClock}(P_i, m_i)[P_k] \leq \text{VectorClock}(P_j, m_j))[P_k] \tag{4}$$

- Vector clocks are not always comparable and we can also compare a vector clock to a Delivered-array. We define

$$\text{VectorClock} = \max(\text{VectorClock}(P_i, m_i), \text{VectorClock}(P_j, m_j)) \tag{5}$$

to mean that

$$\forall P_k(\text{VectorClock}[P_k] = \max(\text{VectorClock}(P_j, m_j)[P_k], \text{VectorClock}(P_i, m_i)[P_k])) \tag{6}$$

- For at vector clock VectorClock let VectorClock$+P_j$ be the same vector clock but where 1 is added to the position VectorClock$[P_j]$
- The following is a vector clock based protocol for consistent communication

  1. P_i: Initially let VectorClock$(P_i)[P_j] = 0$ and let Delivered$(P_i)[P_j]$ for all P_j
  2. On input (P_i, m) at P_i
     (a) Let VectorClock$(P_i)[P_i]$ = VectorClock$(P_i)[P_i] + 1$
     (b) Let VectorClock(P_i, m) = VectorClock(P_i)
     (c) Send (P_i, m) along VectorClock($\mathrm{P_{i,m}}$)
  3. On receiving (P_j, m) at P_i along with VectorClock(P_j, m)
     (a) Wait until .VectorClock$(P_j, m) \leq$ Delivered$(P_i) + P_j$.
     (b) Then deliver (P_j, m)
     (c) Let VectorClock(P_i) = max(VectorClock(P_i, m_i), VectorClock(P_j, m_j))
     (d) Increment Delivered$(P_i)[P_j]$ by 1

5.4 Total Order

- **Total Order:** If a correct P_k delivered (P_i, m) and then later delivered (P_j, m') then it holds for all correct P_m that if they deliver (P_k, m'), then they earlier delivered (P_i, m)
  - To ensure total order you can use the casual order system and ping all other machines an wait for an ack for all other machines, this ensures that there are no

6 Confidentiality (5)

6.1 Confidentiality, Secret-Key Systems

6.1.1 General

- A secret-key cryptosystem consists of three algorithms
  - One for generating a key: G
  - One for encryption: E
  - One for decryption: D
- The algorithm G typically generates a key by outputting a randomly chosen bit stïrng of a fixed length
- The algorithm E takes as input a key k and a message m and outputs a **ciphertext** c
- The algorithm D takes a ciphertext c and a key k and produces a plaintext $D_k(c)$
- It must hold that

$$m = D_k(E_k(m)) \tag{7}$$

- The system should makes sure that when an adversary sees c but not k it should have no idea whatsoever what it represents

6.1.2 The One-time Pad and Perfect Secrecy

- If each encryption key is only used once, it is not too difficult to design an unbreakable encryption algorithm, which takes the form of the so-called onetime pad.

- A one-time pad is quite useless in practice

- The **one-time pad** is constructed as follows
  - Assuming the message is a string of bits labeled $m_1, \ldots, m_t$
  - The key will by a random bit sting of the same length as the message labeled $k_1, \ldots, k_t$
  - Encryption is done by taking the bit-wise xor of the strings
    - The ciphertext $c_1, \ldots, c_t$ is defined by $c_i = m_i \oplus k_i$
  - The receiver who knows the same key can recover the plaintext by decrypting each i'th bit as

$$c_i \oplus k_i = m_i \tag{8}$$

- **Theorem 5.1** When one-time pad is used for encryption the ciphertext is always a uniformly distributed bit string, in particular, it is independent of the plaintext.

- It is said that a cryptosystem has **perfect secrecy** when the cipher text is independent of the plaintext
  - This holds no matter how much computing power the eavesdropper has

- **Theorem 5.2** Suppose a cryptosystem can handle $\mathcal{M}$ possible plaintexts, and uses $\mathcal{C}$ ciphertexts and $\mathcal{K}$ keys. If the system has perfect secrecy, then it must be the case that $K \geq C \geq M$

6.1.3 Practical systems, Definition of Security

- In practical systems one typically have to settle for computational security
  - One would have to spend unrealistically amount of time to break the system

- Computational security has several consequences for how we should design cryptosystems and use them
  - The adversary should have no idea what we are sending even if we use the system several times,

- Good encryption algorithms work not only with the input k, m as input, they also make a choice of some variable who value changes from one encryption operation to the next
  - Is called a **nonce**
  - It must chosen in some way such that we will not use the same value twice
  - It can take the form of a counter or random bits
  - This makes sure that if we encrypt the same message twice it will have different cipher texts
  - To emphasize that a nonce n was used in the encryption process we write

$$c = E_k(m, n) \tag{9}$$

- **Definition 5.3** Consider any adversary who plays the above game and whose computing power is limited in the sense that whatever algorithm he runs terminates in time much less than the time it takes to try all possible keys in the cryptosystem. No such adversary can guess whether he is in case 1) or 2) (with probability better than essentially a random guess). Cases where m is a message and r is a random message the same length as m
  1. $C = E_k(m, n)$
  2. $E_k(r, n)$

6.1.4 Exhaustive Search

- A consequence of reusing the same many times is that a system can only be secure if the adversary's resources are limited
  - We have to assume that an adversary does not have enough computing power to run through all possibilities for the key k
- If the hacker finds out one or more plaintext(s) m corresponding to a ciphertext(s) c. It means he can execute the simple algorithm **exhaustive key search**:
  1. Initialize an empty list L
  2. For every possible key k': compute $D'_k(c)$ and check if $D'_k(c) = m$ for all the plaintext/ciphertext pairs (m, c) that are known

- If this is the case add k' to the list L

- **Fact:** If the adversary knows u bits of the plaintext (and matching ciphertext), and u is larger that the length of the key, then we can expect that exhaustive search will identify the correct key.

- To make sure that exhaustive search is infeasible, a secure system these days must use keys of length about 128 bits or more – since doing 2^{128} repetitions of some non-trivial computation is currently considered completely infeasible.

  - Key length by itself is no guarantee for security it is only a necessary condition, because given m, c there might be a much faster way to find the key than to try all possibilities

- Thinking in terms of exhaustive search is part of estimating the `cost of an attack` and balancing that with the `probability of an attack`.

  1. Say that if someone breaks your system you will lose C dollars
  2. There is a probability p that an attacker can break into the system
  3. Then the expected cost of attacks on the system is pC dollars.
     - You in general want the amount pC to be negligible i.e. something you don't mind paying

6.1.5 Stream Ciphers

- A steam cipher is an algorithm G that expands a short key k and a nonce n to a much longer random looking string $G(k, n)$ which is then used to encrypt the message as if it was a onetime pad

  - The receiver must know not only k but also n to decrypt the message,
  - n is not secret so it can just be send along the ciphertext

- A stream cipher in practise does not generate the entire output at once and it does not know the length in advance

  - It fits nicely with application where the input to be encrypted arrives as a stream e.g. byte by byte
    - e.g. a user typing on a keyboard

- A well known example of a stream cipher is RC4 which is often used in web browsers, which is rather unfortunate as it has some known weaknesses and should not be used.
  - However, in recent years other, more and very fast secure stream-ciphers have been proposed such as `SALTA20` and `SNOW`

6.1.6 Block Ciphers

- Block ciphers encrypt in their basic form a fixed size block of data, and output a block of the same size as the input.
  - Examples are the former US standards DES (56 bit keys, 64 bit blocks), triple-DES (112 bit keys, 64 bit blocks) and the present standard AES (128 bit keys and blocks).
- To use a block cipher in practice, one needs so called Modes of Operation
  - Which are general methods that allow using a block cipher to encrypt a string of data of any length, and also to achieve security as required in Definition 5.3.
  - Examples are Cipher Block Chaining (CBC) mode, Counter (CTR) Mode, and Output Feedback (OFB) mode.
  - The modes need as input not only the key and the message, but also a nonce
    - In modes-of-operation lingo the nonce if called an **initialization vector**, IV
- **OFB mode** is a mode that can be used to make a block cipher function in a way similar to a stream
  - The output is computed by feeding the output block back into the encryption function
  - It creates a seemingly random stream of bits as needed for a stream cipher
- A commonly used mode is **CBC mode**
  - Assume the message consists of 128 bit block $M_1, \ldots, M_t$ where we pad the last block i in some way if it does not fill the required block length

- Then the cipher text will be $t+1$ blocks $C_0, \ldots, C_t$, where $C_0 = IV$ and for $i = 1, \ldots, t$

$$C_i = AES_K(M_i \oplus C_{i-1}) \tag{10}$$

- **CTR mode**
  - The message consists of 128 bit block $M_1, \ldots, M_t$ as in CBC mode and the cipher text depends on an IV
  - The ciphertext will be $t+1$ block $C_0, \ldots, C_t$ where $C_0 = IV$ and for $i = 1, \ldots, t$

$$C_i = AES_k(IV + i) \oplus M_i \tag{11}$$

- $IV + i$ means think of IV as a 128 bit number and add i to this number modulo 2^{128}
- CTR mode can compute multiple blocks in parallel
- As a rule of thumb, one will need
  - a good block cipher
  - an appropriate mode of operation
  - a reasonable way to choose initialization vectors to get a useful encryption scheme.
- The basic problem with secret-key systems is that one must have the key in place at both the receiver and sender before sending data

6.2 Confidentiality, Public-Key Systems

6.2.1 General

- A public-key cryptosystem also has three algorithms G, E, D for key generation, encryption and decryption
  - A public-key system makes used of a pair of matching key, a *public key pk and a /secret key sk*
  - The user A will generate a pair of keys in private on his own machine and then publish *pk* but keep *sk* private
  - It must be the case that even though there is a connection between, it must be a difficult problem to compute *sk* from *pk*

- Therefore anyone can encrypt a message m intended for A by running E on input m and get $c = E_{pk}(m)$. and A can then decrypt this by running D on input sk, c because the system which means we have

$$m = D_{sk}(E_{pk}(m)) \tag{12}$$

- The same message must encrypt to something different every time, since if it is not the case the adversary could win the following way:

  1. The adversary submits a message m to the oracle
  2. The adversary receives a ciphertext c from the oracle
  3. The adversary encrypts $c' = E_{pk}(m)$ and concludes that c is an encryption of m if $c' = c$ and an encryption of a random r otherwise.

- Public-key cryptosystems can also be broken using exhaustive search
  - Usually, there is only one private key corresponding to a given public key, so the adversary can just try all possible secret keys until he finds the one that matches
  - However, for all known public key systems, there are algorithms known for computing sk from pk that are much faster than just trying all possibilities.
    - Therefore the size of keys for public-key systems are usually much bigger than they are for secret-key systems.

6.2.2 RSA

- RSA is a public key system
  - The public key consists of two numbers n and e
  - The private key consists of number n and d
  - The number n is called the **modulus** an is the product of two prime numbers p, q
  - The numbers e and d are chosen to satisfy a particular relation that involves in an essential way the prime factor p, q
    - One computes $d = f(e, p, q)$ for some easy to compute function f

    * It can be shown that computing the private key from the public one is as hard as solving the problem of finding p, q from n (*factoring problem*)

- The basic version of RSA which is deterministic and therefore insecure is as follows
  - Messages are numbers in the interval $[0 \dots n-1]$ and two encrypt a number m one computes $c = m^e \mod n$
  - One decrypts a message by computing $c^d mod n$ and the special choice of e, d ensures that we always have

$$c^d \mod n = (m^e \mod n)^d \mod n = m \tag{13}$$

- The only way to break this system is the possibility of factoring n
  - The best known algorithms for factoring are capable of factoring a \$k\$-bit RSA modulus in time $2^{O(\sqrt[3]{k})}$
  - To make sure one cannot use these algorithm in reasonable time RSA keys must have size at least 2000-3000 bits
    * This size is not fixed and must increase with improvements in hardware and the increasing

- To compute d from e, p and q the following is done
  - e must be chosen such that the greatest common divisor of e and $(p-1)(q-1)$ is 1
  - Then one computes d such that $d \mod (p-1)(q-1) = 1$.
  - The condition on e ensures that a suitable d always exists and is usually written as follows:

$$d = e^{-1} \mod (p-1)(q-1) \tag{14}$$

6.2.3 Using Public-Key Systems in Practice

- What one usually does is therefore to use the public key system just once in a session to communicate a key for a secret-key system, which is then used on the actual data.
  - It is sometimes known as "key enveloping"

  - Done since public key systems are usually much slower than secret key systems

- RSA Encryption with OAEP is a secure way to do padding done the following way

  1. To encrypt a key k, one first computes a padded version $OAEP(k, R)$, where R is a string of random bits.
     - The length of R is chosen as a function of the size of the RSA modulus such that $OAEP(k, R)$ is a number of suitable size for encryption under RSA.
  2. The ciphertext is $OAEP(k, R)^e \mod n$.
  3. The receiver computes $(OAEP(k, R)^e \mod n)^d \mod n = OAEP(k, R)$, checks that the format of the result is correct and if so, the receiver recovers k from $OAEP(k, R)$.
     - The OAEP function is constructed such that this is easy.

- Public key schemes, including RSA, should never be used in practice without use of a secure padding method such as OAEP

7 Authenticity (6)

7.1 Authenticity, Secret-Key Systems

7.1.1 General

- A secret-key system for authenticity consists of three algorithms G, MAC, V
  - G generates a key
    - Produces a key by just outputting a random string of fixed length
  - MAC, is used to authenticate a message
    - Stands for Message Authentication code
    - Takes as input a message m and a key k and output a MAC, $c = MAC_k(m)$
  - V is used to verify a received message
    - m and c is send to the receiver, who will run V on the input
    - The result of $V_k(m, c)$ is either *accept* or *reject*

- An authentication scheme must have the property that if no one tried to modify the message, the receiver will accepts, i.e. the following must be true

$$V_k(m, MAC_k(m)) = accept \tag{15}$$

- For security the adversary is allowed
  - to specify any number of messages $m_1, \ldots, m_t$ and he is given valid MACs $c_1, \ldots, c_t$ for these messages
  - to specify pairs of form m, c and will be told if c is a valid MAC on m
- **Definition 6.1** Consider any adversary who runs in time much less than what exhaustive key search would take. The authentication scheme is secure if no such adversary can play the game specified above and in the end produce a message m_0 and a MAC c_0 such that $V_k(m_0, c_0) = accept$ and $m_0 \notin \{m_1, \ldots, m_t\}$
- There is no requirement that the MAC algorithm should be randomized

7.1.2 Unconditional Authentication

- An unconditional secure way to do message authentication are for the sender and receiver to agree in advance on a table containing for each message a randomly independently chosen t-bit mac
  - The table functions as a key
  - The adversary cannot find the correct MAC for a message except with probability 2^{-t}
  - It has the property that the key get larger the more possible messages we have

7.1.3 Practical systems and exhaustive search

- In real life we need systems where we can use a small, fixed key
  - The should be much fewer possibilities for the key than there are possible messages
  - Means that an adversary who has seen a few valid messages and MACs that the key can be found by exhaustive search

        * He runs through all possibilities and generates MAC's for all messages that were actually sent
        * If all these MAC's are identical to those that were sent by the legitimate sender, the adversary assumes he found the correct key.
        * To ensure that such an exhaustive search is infeasible, the same constraints on key (≥ 128) holds
    - The MAC itself cannot be too short
        * Otherwise an adversary might be able to simply guess a MAC for a falsified message
        * MAC's of 64 bits are used

7.1.4 Example MAC algorithms

- The construction known as the **CBC-MAC** builds a MAC algorithm from any secure block-cipher
    - CBC stands for cipher block chaining
    - To computer a MAC one simply encrypts the message in CBC mode and defines the MAC to be the last block of the ciphertext
    - A MAC can be verified simply by recomputing the MAC from the received message and comparing to the received MAC
    - Since the last block of the CBC cipher text depends on both the key and the entire message, any change to the message would result in a completely different last black
- The construction known as **HMAC** builds a secure MAC from any secure cryptographic hash function
    - The **SHA1** hash function is used
    - The hash function is efficient to compute and produces a fixed size output, but complicated enough that it is hard to invert.
    - MAC on message m and key K is just $SHA1(m||K)$
        * $m||K$ means m concatenated by K.
        * The actual construction applies some complicated steps to obtain m and K the string that is actual the input to SHA1
- MAC algorithms are generally as fast as secret-key encryption, but suffer of course from the same key distribution problem that exist for

any secret-key construction: we must have the secret key in place at sender and receiver before we can send anything.

7.2 Authenticity, Public-Key Systems

7.2.1 General

- A public-key system for authenticity consists of three algorithms G, S, V
  - S is used to authenticate (sign) a message
  - V is used to verify the received message
    - Must have the same key setup as in public-key encryption
  - G is more complicate than for MAC schemes
    - The output must be a pair of keys with the right relation between them
- A can send an authenticated message m by computing $c = S_{sk}(m)$ and send m, c
  - The receiver who (like everyone) knows A's public key pk can run V to get the result $V_{pk}(m, c)$ which is either an accept or reject
- For any message m and matching keys pk, sk we have

$$V_{pk}(m, S_{sk}(m)) = accept \tag{16}$$

- **Definition 6.2** The adversary is given the public key pk. He is allowed to specify any number of messages he wants, say $m_1, \ldots, m_t$ and he is given valid authenticators $c_1 = S_{sk}(m1), \ldots, c_t = S_{sk}(m_t)$ for these messages. The public-key authentication scheme is secure if the adversary cannot efficiently produce a message m_0 and an authenticator c_0 such that $V_{pk}(m_0, c_0) = accept$, and $m_0 \notin \{m_1, \ldots, m_t\}$
- AA system satisfying definition 6.2 is said to be **unforgeable under chosen message attack**.

7.2.2 Security of public-key systems and difference to secret-key

- Because the adversary can find the key must faster than exhaustive when using public key systems the key needs in general to be much larger for public-key authentication than for the secret-key case.

- Everyone can check if the message is send from A
  - Therefore they are also called *digital signature schemes*
- For public-key encryption, it is important that the receiver uses the right public key when checking a signature, otherwise he can be made to belive that a message comes from A when this is not the case.

7.2.3 Examples of Digital Signature systems

- It is usually the case that given some public-key crypto-system, the underlying techniques can also be used to build public-key signature schemes, though we usually need additional tools to get secure schemes.
- A (insecure) attempt to use RSA with public key $pk = (n, e)$ and private key $sk = n, d$ to generate signatures works as follows
  - Assume that the message is a number m in $[0 \dots n-1]$.
  - The signer who is the only one who knows the private key will apply the private-key operation to the message.
    - The signature on m is $S_{sk}(m) = m^d \bmod n$.
  - The special way e and d are chosen will ensure that we have

$$s^e \bmod n = (m^d \bmod n)^e \bmod n = m \tag{17}$$

- When you receive a pair m, s, you can check the signature by verifying whether $s^e \bmod n = m$ if the condition is satisfied $V_{pk}(m, s)$ outputs accept and reject otherwise
- In general, the signature schemes in the simplistic form mentioned here are the same speed as the corresponding crypto-systems.
  - For RSA, if we apply the optimization where we use a small number as e, then checking a signature will be much faster than generating one.
  - It is important in applications where a signature is generated once, but verified many times.

7.2.4 The problem with the simplistic scheme

- The given use of RSA does NOT satisfy the definition of security
  - An adversary could choose some s, compute $m = s^e \bmod n$ and claim that m is a signed message
  - m might not be meaningful at all

7.2.5 Hash Functions

- The solution to both the speed and security problems is known as **cryptographic hash functions**, a such function h should have the following properties:
  - It should be able to take a message of any length as input
  - It should produce an output of fixed length
  - It should be fast to compute speed similar to the best secret-key systems
  - It should be a hard computational problem to produce a **collision**:
    - Two inputs x, y such that $y \neq y$ but $h(x) = h(y)$
    - This means that to such x, y exists but are hard to find
- To make an attack trying to produce a collision infeasible good hash functions must have output length at lease 160 bits

7.2.6 Hash-and-Sign Signatures

- Signing $h(m)$ is just as convincing as signing m itself therefore
- To fix a hash function h to be used by all users one uses a signature scheme such as basic RSA with signature algorithm S and defines a new signature scheme with signature algorithm S'
  - The signature on message m is defined to be $S'_{sk}(m) = S_{sk}(h(m))$
  - The new verification V' on message m and signature s $V'_{pk}(m, s)$ will compute $h(m)$ execute $V_{pk}(h(m), s)$ and accept if and only if V said accept
- Since $h(m)$ is usually much smaller than the message the given and therefore fixes the speed issues

- The attack on simplistic RSA does not work if messages are hashed before they are signed
  - The adversary has to choose some s and m such that $h(m) = s^e \bmod n$, which is the problem of inverting h

7.2.7 Replay attacks

- The secret-key and public-key approach to authenticating messages are actually not satisfactory by themselves in all cases
  - The reason for this is that if one receive a message m with a digital signature from A, this only proves that at some point, A produced this message
  - It leaves open for a **replay attack**
    - Where an adversary take a copy of the signed and send as many time as he wants
- We want a real authentic channel where B receives the exact same sequence of messages that A send
  - Or to come as close as we can
  - One trivial way to ensure this is have the sender make sure that he never sends exactly the same message twice.
    - For instance by appending a sequence number, and also add a MAC (computed over both message and sequence number.
    - Then we can have the receiver store every message he ever receives (or at least the sequence numbers).
      - This will allow the receiver to filter out every replayed message and correctly replay the messages in order
      - Of course, this is hardly a practical solution
      - Even if we do use sequence numbers
  - It can also be achieved by saving the sequence number N of the latest received message
    - Leaves the possibility open to reject to late received messages
  - Another way is to use timestamps and save all received timestamps
    - helps against replay attacks but

    - A better way is to check that the timestamp is no to old compared to the local machine time
  - One may use interaction
    1. The receiver first send a number R to the sender
       - This can be chosen at random or a sequence number
       - The only requirement is that it hasn't been used before
    2. The sender sends the message plus a MAC computed over the message and R
       - This will prevent replay and there is no need for synchronization

7.2.8 Getting both Confidentiality and Authenticity

- Pitfalls when using a combination of the two types of techniques to achieve both confidentiality and authenticity
  - In the secret-key case, we will want to compute MAC's and also encrypt messages.
    - One must use different and independent keys for the two purposes
    - There is NO guarantee for security if the same key is used.
- There exists specially engineered encryption modes that provide both confidentiality and authenticity at the same time, using one key.
  - Such encryption schemes are known as authenticated encryption
  - It is simpler to use a single scheme that provides both confidentiality and authenticity than having to use two different tools at the same time.
  - Examples of authenticated encryption schemes are OCB, GCM and CCM.
- Therefore, it is often recommended to first encrypt and then compute a MAC on the ciphertext.
  - Otherwise some information about the message m may be revealed in the MAC

8 Synchronous Agreement (7)

8.1 Clock Synchronisation

8.1.1 General

- Computing devices can synchronize their clock by interacting with more accurate clocks which they are connected to via a computing network
  - A few atomic clocks can help the computing devices equipped with low-accuracy quarts clock not drifting too much

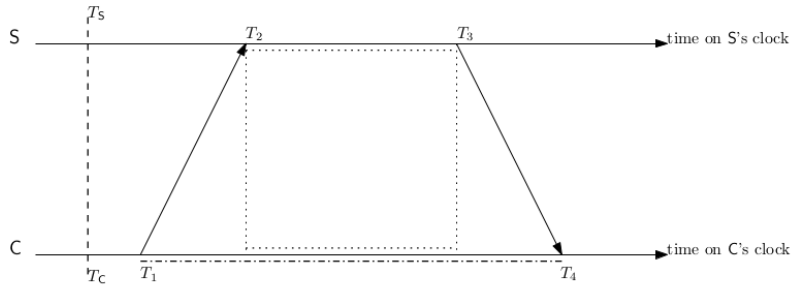
8.1.2 GPS Clock Synchronisation

- The global positioning system consists of a large number of satellites
  - Each of them are equipped with an atomic clock and they constantly transmit their position and their time
    - Their position consists of three spatial coordinates (x, y, z)
    - The time consists of one coordinate t
  - It is broadcasted public and can received by anyone with a GPS receiver
  - If one receive for signals $(x_1, y_1, z_1, t_1), \ldots, (x_4, y_4, z_4, t_4)$ one can compute their own position and time
    - Is comparable to solving four equations with four unknowns

8.1.3 NTP Clock Synchronisation

- The details of the Network Time Protocol
  - A server S which have an atomic clock or another precise clock
  - A client C with a drifting clock which occasionally needs to be synchronised
  - The algorithm works under two assumptions
    1. During the time it takes to run the protocol, the clocks of the server and the client only drift negligibly apart

2. The time it takes to send a package from client to server is the same as it takes to send from server to client



- When a client wants to synchronize its clock the following protocol is executed

  1. C sends a time request message to S and stores its current system time T_1
  2. Upon receiving the time request message, the server S stores its current system time T_2
  3. The server S prepares a time response message which includes the time T_2. Right before sending the response to the client C, the server S measure its current system time T_3 again and includes its in the message
  4. Upon receiving the message (time response, T_2, T_3), the client C messures its current system time T_4. It can now compute two numbers:

$$\begin{aligned} \text{TransEst} &= \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \\ \text{OffsetEst} &= (T_1) + \text{TransEst} - T_2 \end{aligned} \tag{18}$$

- and adjusts its own clock using this number
- The client often speeds up the clock a bit to set the clock correct since sudden jumps forward or backward in time can cause problems
- To accommodate the problem of potential errors when transferring the protocol the client typically runs the protocol several times and takes the execution where TransEst was lowest

- To make sure that assumption 2 is probably true
- The rational is that when transfer errors happens the transmission times tend to be higher

8.2 Round-Based Protocols

- A **fully synchronous round-based model** in the simplest instantiation
  - It has n parties or process called $P_1, \ldots, P_n$.
  - The protocol π proceeds in rounds
  - The idea is that in each round each process is allowed to send a message to each other process
    - Send nothing is an option which is denoted NOMSG
  - In this model it is assumed that all processes have perfectly synchronized clock and transmission time is fixed
  - It is hardly realistic
  - Though even if clocks are not perfectly synchronized we can still hope to set bounds on clock drift and transmission time and predict when a message ought to arrive
    - If a client expects that a server will send a message at time t
    - It knows that its offset from the servers clock is at most Offset
    - It knows that it takes at most Trans seconds to send a message
    - Then it can conclude that no message was send if nothing was received at time $t + 2\text{Offset} + \text{Trans}$
- Generalizing this idea to a protocol π consisting of many rounds
  - The bounds assumed is
    - The computation that needed to be done in each takes the same time in each round, which is denoted MaxComp

    - The positive time bound MaxTrans on how long it maximally takes to send a message between two correct process
    - The positive time bound MaxDrift on how long any correct process drifts from real time is assumed

- The SlotLength can be computed as SlotLength = 2MaxDrift + MaxTrans + MaxComp
    - The time t_0 is the time which all parties start running the protocol
    - Each round runs within the a time slot
    - Rounds are index by natural numbers r
    - Let $\text{SlotBegin}^r = t_0 + r \cdot \text{SlotLength}$
    - Let $\text{SlotEnd}^r = t_0 + (r+1) \cdot \text{SlotLength}$
    - Round r is assigned the time slot $\text{Slot}^r = [\text{SlotBegin}^r, \text{SlotEnd}^r)$

- A round based protocol proceeds as follows
    1. Each P_i starts round $r = 0$ at time $\text{SlotBegin}^0 = t_0$
        - At time SlotEnd^0 the computation of round 0 should have terminated
        - Let $(m^0_{i,1}, \ldots m^0_{i,n})$ be the messages to be send
        - For each P_j, send $(\text{MSG}, 0, m^0_{i,j})$ to P_j
    2. In rounds $r = 1, 2, \ldots$, party P_i runs as follows:
        (a) On message from $(\text{MSG}, r-1, m)$ from P_j after timer $\text{SlotBegin}^r - 2\text{MaxDrift}$ and before $\text{SlotBegin}^r + \text{MaxTrans} + 2\text{MaxDrift}$ if this the first message of the form $(\text{MSG}, r-1, \cdot)$, then store $m^{r-1}_{j,i} = m$
        (b) At time $\text{SlotBegin}^r + \text{MaxTrans} + 2\text{MaxDrift}$, for each P_j where no $m^{r-1}_{j,i}$ is stored
            - Define $m^{r-1}_{j,i} = \text{NOMSG}$
            - Using as input the values $(m^{r-1}_{1,i}, \ldots m^{r-1}_{n,i})$ now stored and defined, perform whatever computation π prescribes for round r
        (c) At time $\text{SlotBegin}^r + \text{MaxTrans} + 2\text{MaxDrift} + \text{MaxComp}$
            - The computation of round r should have terminated
            - Let $(m^r_{i,1}, \ldots m^r_{i,n})$ be the messages to be send

        * For each P_j, send \$(MSG, r, $\mathrm{m}_{\mathrm{i,j}}^{\mathrm{r}}$)\$ to P_j.

    - If in round $r-1$ a correct P_j send $(\mathrm{MSG}, r-1, m)$ to P_i before $\mathrm{SlotBegin}^{r-1} + \mathrm{MaxTrans} + 2\mathrm{MaxDrift} + \mathrm{MaxComp}$ and the message was received too late by P_i such that $m_{j,i}^{r-1} = \mathrm{NoMsg}$ in round r it is said that the correct message was dropped by timeout

        * If the assumptions on the timebounds are all correct this will never happen

8.3 Unscheduled Consensus Broadcast using Signatures

- In **consensus broadcast**, some broadcaster sends a message to all other parties

    - It must be guaranteed that all correct parties receive the same message, even if the broadcaster and some of the receivers are corrupted

    - Sometime called just **broadcast** or **consensus**

- **Scheduled broadcast** is a type consensus broadcast

    - The broadcast happens in a planned round

    - All parties known that it will happen at the specified round

- **Unscheduled broadcast** is a another type consensus broadcast

    - The broadcast can be initiated by the broadcaster in any round

- **Unscheduled Consensus Broadcast** is a protocol between n parties $P_1, \ldots, P_m$

    - A party P_i can get inputs of the form (Send, m)

        * If it is correct, it get this input at most one for each m

    - A party P_j can give outputs of the form $(\mathrm{Deliver}, P_i, m)$.

    - The following properties are important

        * **Validity:** If a correct P_j outputs $(\text{Deliver}, P_i, m)$, and P_i is correct, then at some point P_i got the input (Send, m)
        * **Agreement:** If a correct P_j outputs $(\text{Deliver}, P_i, m)$, then eventually every correct P_j outputs $(\text{Deliver}, P_i, m)$
        * **Timing:**
            · If a correct P_i gets input (Send, m) then all correct P_j output $(\text{Deliver}, P_i, m)$ in the next round.
            · If a correct P_j outputs $(\text{Deliver}, P_i, m)$ in round r_j and a correct P_k outputs $(Deliver, P_i, m)$ in round r_k, then $|r_k - r_j| \leq 1$.
    - The Validity and Agreement properties are safety properties
    - The Timing property is neither a safety property or a liveness property
    - Uses digital signatures
        * Each party P_i has a publicly known verification key vk_i for a signature chee
        * Only P_i knows the signing key sk_i
        * It is important that all process agrees on the verification keys vk_i for all process
            · It is assumed that the round based protocol have a initialization round where all parties broadcast the public keys.
            · In practice there are no such round

- It is assumed that all round based protocol run the following code in round 0:
    - **initialize** In round 0 each party P_i samples a key-pair $(\text{vk}_i, \text{sk}_i) \rightarrow \text{Gen}(1^k)$, and broadcasts vk_i
        * k is the security parameter
        * These keys are delivered in one round
        * All parties in the futher use vk_i as the verification key of P_i

- The unscheduled round based protocol is as follows

- **send** P_i: On input (Send, m), compute $\sigma_i = \text{Sig}_{\text{sk}_i}(Send, m)$ and send $(\text{Send}, m, P_i, \sigma_i)$ to all parties
- **deliver** P_j: On input $(\text{Send}, m, P_i, \sigma_i)$, if $(\text{Deliver}, P_i, m)$ was not output before, output $(\text{Deliver}, P_i, m)$ and send $(\text{Send}, m, P_i, \sigma_i)$ to all parties

8.4 Consensus Broadcast using Authenticated Channels

- In the **Authenticated Channels model** when P_i receives message m from P_j, he knows that m really came from P_j
  - If P_i is corrupt he may later choose to lie and claim that the got some other message from P_j
  - MACs are used between each pair of parties
- A protocol for Scheduled Consensus Broadcast starts in a particular globally known round and takes place between n parties $P_1, \ldots, P_n$
  - There is a designated party known as the broadcaster
  - P_1 is the broadcaster
  - The input of P_i is a message $m \in \{0,1\}^*$
  - The output of $P_i \neq P_1$ is a result $r_i \in \{0,1\}$
  - For P_1 we define $r_1 = m$
  - At the beginning of the protocol $r_i = \perp$ for all correct processes $P_i \neq P_1$
  - When P_i is ready to give an output it sets r_i to a value in $\{0,1\}^*$ and never changes it again
  - A protocol is said to be executed correctly if all correct processes started running the protocol in the same round
  - The following properties are imposed on correctly executed protocols:
    - **Validity:** If P_1 is correct, then it holds for each correct process $P_i \neq P_1$ that $r_i = m$

    - ∗ **Agreement:** It holds for each pair P_i and P_j of correct processes that are not P_1 that is $r_i \neq \perp$ and $r_j \neq \perp$ then $r_i = m$
    - ∗ **Termination:** There exists some constant cc such that whenever the protocol starts executing in round t_0, then by round $t_0 + c$ it holds for all correct P_i that $r_i \neq \perp$
  - Termination and Agreement should hold even if P_1 is not valid
  - All properties must hold even if some of the process are not valid

- The protocol for **Scheduled Consensus Broadcast** which works four parties of which at most one cat be Byzantine

  1. In round 1 party P_1 sends m to P_2, P_3, P_4
  2. Let m_i be the message received by P in round 1. In round two party P_i sends m_i to P_2, P_3, P_4.
  3. Let $m_{j,i}$ be the message received by P_i from $P_j \neq P_1$ in round 2.
     - By definition let $m_i, i = m_i$
  4. In round 3 party $P_i \neq P_1$ does the following:
     - If there is a message h such that h occurs twice in $(m_{2,i}, m_{3,i}, m_{4,i})$ then let $r_i = h$
     - Otherwise, set r_i to a special error message and then halt.s

8.5 Byzantine Agreement using Authenticated Channels

- The Byzantine Agreement is a protocol between n parties, which is denoted as $P_1, \ldots P_n$
  - The input of P_i is a vote $v_i \in \{0, 1\}$
  - The output of P_i is a result $r_i \in \{0, 1\}$
    - ∗ At the beginning of the protocol we set $r_i = \perp$ for all correct processes
  - When P_i is ready to give an output it sets r_i to 0 or 1 and then never changes it again

- The protocol was executed correctly if all the correct process started running the protocol the same round
- The following properties are imposed on correctly executed protocols
  - **Validity:** It holds for each correct P_i that if $r_i \neq \perp$ then there exists a correct process P_j such that $r_j = v_j$
  - **Agreement:** It holds for each pair P_i and P_j of correct processes that if $r_i \neq \perp$ and $r_j \neq \perp$ then $r_i = r_j$
  - **Termination:** There exists some constant c such that whenever the protocol starts executing in round t_0, then by round $t_0 + c$ it holds for all correct P_i that $r_i \neq \perp$
- To work with Byzantine agreement one must always assume that $t < n/2$ byzantine corrupted

8.6 Dolev-Strong: Scheduled Broadcast using Signatures

- A solution which works for any n and any $t < n$
- P_1 is the broadcaster
- The protocol makes use of variables of form $\text{Relayed}_i(m)$
  - This flag signals whether P_i has seen a signature from the broadcaster $\$P_1\$$on message m and has relayed it to the other player
- **initialize** In round 0 party P_i samples a key-pair $(vk_i, sk_i) \leftarrow \text{Gen}(1^k)$ where k is the security parameter
  - All parties P_i set $\text{Relayed}_i(m) = \perp$ for all possible message m from the message domain
- **broadcast**
  - On input m in round 1
    - Party P_1 computes $\sigma_1 \leftarrow \text{Sig}_{sk_1}(m)$ and sends $(m, \{\sigma_1\})$ to all parties.
    - Set $r_1 = m$, set $\text{Relayed}_1(m) = \top$ and halt

- In round r, if Pi receives a message of form (m, Σ) where Σ is a set of signatures and if $\text{Relayed}_i(m) = \perp$ proceed as follows:
  - Call Σ valid for m in round r if it contains signatures σ_j from $r-1$ distinct parties P_j such that $\text{Vec}_{vk_j}(m, \sigma_j) = accept$
  - One of the parties have to be P_1
  - If Σ is valid for m in round r, then compute $\sigma_i \leftarrow \text{Sig}_{sk_i}(m)$, let $\Sigma' \leftarrow \sigma \cup \{\sigma_i\}$ and send (m, σ') to all parties. Then set $\text{Relayed}_i(m) = \top$
- In round $n+2$ party P_i computes its output as follows:
  - If there is exactly one message m such that $\text{Relayed}_i(m) = \top$, then set $r_i = m$
  - Otherwise, set $r_i = NoMsg$

8.7 Lower Bound on Round-Complexity of Broadcast using Signatures

- To tolerate $t < n$, then one can make a protocol which only runs in $t+1$ rounds fulfilling these conditions
  1. After some round of initialisation, the protocol only uses point to point communication
  2. Ignoring the probability that the signature schemes could be broken the protocol is perfect
     - i.e. validity,termination and agreement hold with probability 1

Theorem 7.1 Consider a deterministic protocol for n parties where all parties terminate in round r. Assume that up to t parties might be crash-silent corrupted and that $n \geq t+2$. If the system achieves weak Byzantine agreement, then $r \geq t+1$

9 Asynchronous Agreement (8)

9.1 Byzantine Asynchronous Broadcast using Authenticated Channels

- A Byzantine broadcast protocol for the full connected model with authenticated channels and asynchronous communication with eventual

delivery

- There are n parties $P_1, \ldots, P_n$
  - All can act as the bradcaster
  - P_1 is assumed to always be the broadcaster
  - The broadcaster can get input of the form (BROADCAST, P_1, bid, m), where bid is a fresh broadcast identifier
  - Other parties can give outputs of the form (DELIVER, P_1, bid, m')
- The requirements for the protocol:

Validity 1: If a correct P_i outputs (DELIVER, P_1, bid, m) and P_1 is correct, then at some earlier point P_1 got the input (BROADCAST, P_1, bid, m).

Agreement: If a correct P_i outputs (DELIVER, P_1, bid, m), then eventually all correct P_j output (DELIVER, P_1, bid, m).

Validity 2: If P_1 is correct and gets input (BROADCAST, P_1, bid, m), then eventually all correct P_j output (DELIVER, P_1, bid, m).

- Let n be the number of parties and let $t < n/3$ be the number of Byzantine corruptions to tolerate. A protocol for this is know as Bracha broadcast:

Send P_1: On input (BROADCAST, P_1, bid, m), send (SEND, P_1, bid, m) to all parties.

Echo P_i: On message (SEND, P_1, bid, m) from P_1, send (ECHO, P_1, bid, m) to all parties.

Ready 1 P_i: Once message (ECHO, P_1, bid, m) has been received from $n-t$ parties, send (READY, P_1, bid, m) to all parties.

Ready 2 P_i: Once message (READY, P_1, bid, m) has been received from $t+1$ parties, send (READY, P_1, bid, m) to all parties.

Deliver P_i: Once message (READY, P_1, bid, m) has been received from $n-t$ parties, output (DELIVER, P_1, bid, m) and terminate the protocol.

- Waiting for a message from $n - t$ process gives the process the most information possible without deadlocking
- Waiting for a message from $t + 1$ parties ensures that one hears from at least one correct party
- $n < 3t$ guarantee that there is at least one shared correct party that any two of the correct process heard from
- If P_1 is corrupt and does not send its message, then the correct parties run forever

9.2 Impossibility of Deterministic Asynchronous Byzantine Agreement

- Each Byzantine agreement will be identified by a fresh identifier *baid*
  - Each party can get an input $(\text{VOTE}, baid, v_i)$ where $v_i \in \{0,1\}$ is called the vote
  - Each party can give an output $(\text{DECISION}, baid, d_i)$ $d_i \in \{0,1\}$ is called the decision
- Requirements:

Agreement: If for some *baid* some correct P_i and P_j output $(\textsc{Decision}, baid, d_i)$ and $(\textsc{Decision}, baid, d_j)$, then $d_i = d_j$.
Validity: If for some *baid* some correct P_i gives an output $(\textsc{Decision}, baid, d_i)$, then at some earlier point some correct P_j had the input $(\textsc{Vote}, baid, d_i)$.
Termination: If for some *baid* all correct processes got an input of the form

$(\textsc{Vote}, baid, \cdot)$, then eventually all correct processes give an output of the form $(\textsc{Decision}, baid, \cdot)$.

- It is impossible to make a deterministic protocol which can handle one crash silent error for Byzantine agreement

9.3 Possibility of Randomised Asynchronous Byzantine Agreement

9.3.1 Weak Agreement

- Byzantine agreement in the asynchronous model
  - Each Byzantine agreement ill identified by a fresh identifier *baid*
  - Each party can get an input $(\text{VOTE}, baid, v_i)$ where $v_i \in \{0,1\}$ is called a vote
  - Each party can give an output $\text{DECISION}, baid, d_i$ where $d_i \in \{0,1,?\}$
    - The output ? signals that an agreement could not be reached
    - If some party output 0 then all other parties output 0 or ? and the same for 1

∗ If all parties vote the same no party is allowed to output ?

- Requirements:

Weak Agreement: If for some *baid*, two correct processes P_i and P_j output $(\textsc{Decision}, baid, d_i)$ and $(\textsc{Decision}, baid, d_j)$, where $d_j \neq ?$, then $d_i \in \{?, d_j\}$.

Validity: If for some *baid* some correct P_i gives an output $(\textsc{Decision}, baid, d_i)$, then the following holds:

- If $d_i \neq ?$, then at some earlier point some correct P_j had the input $(\textsc{Vote}, baid, d_i)$.
- If $d_i = ?$, then at some earlier point some correct P_j had the input $(\textsc{Vote}, baid, 0)$ and some correct P_k had the input $(\textsc{Vote}, baid, 1)$.

Termination: If for some *baid* all correct processes got an input of the form $(\textsc{Vote}, baid, \cdot)$, then eventually all correct processes give an output of the form $(\textsc{Decision}, baid, \cdot)$ and will eventually terminate.

- A protocol that works for $n > 5t$ using only authenticated channels

1. All P_i: send v_i to all other parties.
2. All P_i: Wait for v_j from $n - t$ other parties.
   - If $v_j = 0$ for $n - 2t$ of the received votes v_j, then let $d_i = 0$.
   - If $v_j = 1$ for $n - 2t$ of the received votes v_j, then let $d_i = 1$.
   - Otherwise, let $d_i = ?$.

- The protocol has **termination** since there are at most t corrupted parties, so the $n - t > n - 2t$ message in step 2 will eventually arive
- It has **validity** since if all correct parties has the same input v then all parties will receive at least $n - 2t$ votes for d and therefore $d_i = d$

9.3.2 From Weak Agreement to Agreement

- To get Byzantine agreement from Weak Byzantine Agreement the following protocol is considered

1. P_i: Let $v_i^0 = v_i$. Let $r = 0$.
2. P_i: Run Weak Agreement with input v_i^r. Let the output be d_i^r.
3. P_i: If $d_i^r \neq ?$, then set $v_i^{r+1} \leftarrow d_i^r$. Otherwise, let v_i^{r+1} be a uniformly random bit.
4. Let $r \leftarrow r + 1$ and go to step 2.

- A network has **stabilised** on $v \in \{0, 1\}$ ins round r if it holds for all correct P_i that $v_i^r = v$
  - There is a non-zero probability in each round that the network will stabilise

9.3.3 Graded Agreement

- Graded agreements is an extension of weak agreement
- There are two possible outputs for 0, $(0, 1)$ and $(0, 2)$
  - The second component is the grade
  - If someone outputs $(0, 2)$ then all other parties output $(0, 2)$ or $(0, 1)$, similar with $(1, 2)$
  - A decision with grade 2 allows to conclude that all parties made the same decision
  - If the output is ? the grade is always 0
  - Possible outputs: $(0, 2), (0, 1), (?, +), (1, 1), (1, 2)$
- The syntax is as follows
  - Parties can have inputs of the form $(\text{VOTE}, baid, v_i)$ where $baid$ is a fresh BA identifier and $v_i \in \{0, 1\}$ is a vote
  - Parties can have outputs of the form $(\text{DECISION}, baid, d_i, g_i)$, where $d_i \in \{0, ?, 1\}$ is a decision and $g_i \in \{0, 1, 2\}$ is a grade
  - Each possible output is assigned to a number according to the ordered list $n(0, 2) = 1, n(0, 1) = 1, \ldots, n(1, 2) = 5$
- The requirements

Graded Agreement: If for some $baid$ two correct processes P_i and P_j output $(\textsc{Decision}, baid, d_i, g_i)$ and $(\textsc{Decision}, baid, d_j, g_j)$, then $|n(d_i, g_i) - n(d_j, g_j)| \leq 1$.

Validity: If correct process P_i outputs $(\textsc{Decision}, baid, d_i, g_i)$ with $g_i \neq 0$, then at some earlier point some correct P_j had the input $(\textsc{Vote}, baid, d_i)$.

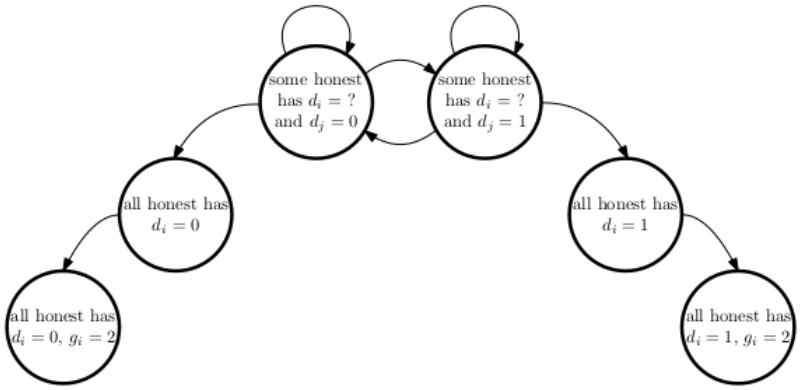
Termination: If for some $baid$ all correct processes got an input of the form $(\textsc{Vote}, baid, \cdot)$, then eventually all correct processes give an output of the form $(\textsc{Decision}, baid, \cdot)$.

Detection: If for some $baid$ and v all correct processes got an input of the form $(\textsc{Vote}, baid, v)$, then eventually all correct processes give an output of the form $(\textsc{Decision}, baid, v, 2)$.

- A protocol which works for $n > 9t$

1. All P_i: Bracha broadcast v_i to all other parties.
2. All P_i: Wait for v_j from $n-t$ parties. Let V_1 be the number of votes for 0. Depending on the interval I that V_1 is in, set the decision and grades as follows:
   - If $V_1 \in [0, t]$, then $d_i = 0$ and $g_i = 2$.
   - If $V_1 \in (t, 3t]$, then $d_i = 0$ and $g_i = 1$.
   - If $V_1 \in (3t, n-4t)$, then $d_i = ?$ and $g_i = 1$.
   - If $V_1 \in [n-4t, n-2t)$, then $d_i = 1$ and $g_i = 1$.
   - If $V_1 \in [n-2t, n-t]$, then $d_i = 1$ and $g_i = 2$.

- The ones with a question ? chooses a new vote $0, 1$ at random



9.3.4 From Graded Agreement to Terminating Agreement

- The termination that one want is typically that all resources allocated to a terminate process is not free
  - Such as in Bracha Broadcast
- It could be dangerous terminating all sub-protocols, since some correct processes P_j might be arbitrarily behind
  - If a process P_i terminates before P_j "wakes up", then P_i will not be around to participate in the sub protocol
- The main trick used is that when honest parties terminate and shut down, they send messages that will later inform slow correct processes about the result of the computation when they wake up

  - They are called **time capsule messages**
  - They are sent before P_i terminates
  - They will be around to be picked up by P_j when it wakes up

- **Definition 8.1 (Asynchronous Termination).** When we say that a party terminates, we mean that it terminates its own process, plus its processes in any subprotocol it started. There is one important exception: it will forever keep running the process that implements the authenticated channels it is using.
  - Could be done by having the operating system of the computer handle the sending and receiving of messages
  - When a correct process terminates it cannot affect the liveness of the protocol it participated in namely the safety property

- The following protocol assumes a protocol for Graded Byzantine Agreement and implements Byzantine Agreement with Asynchronous Terminatation
  - It works if $n > 3t$ and if the Graded BA is secure for the same t and n

1. P_i: Let $v_i^0 = v_i$. Let $r = 0$. Let $\mathsf{GaveOutput} = \bot$.
2. P_i: Run Graded BA with input v_i^r. Let the output be (d_i^r, g_i^r).
3. P_i: If $d_i^r \neq ?$, then set $v_i^{r+1} \leftarrow d_i^r$.
4. P_i: If $d_i^r = ?$, then set v_i^{r+1} to be a random bit.
5. P_i: If $g_i^r = 2$, then output $(\textsc{Decision}, baid, d)$, let $\mathsf{GaveOutput} = \top$ and send $(\textsc{TimeCap}, d)$ to all parties.
6. Let $r \leftarrow r + 1$ and go to step 2.

- I addition to the other rules the parties also run the following extra rules:

Term 1 On receiving $(\textsc{TimeCap}, d)$ from $t + 1$ parties, send $(\textsc{TimeCap}, d)$ to all parties. If $\mathsf{GaveOutput} = \bot$, then set $\mathsf{GaveOutput} = \top$ and output $(\textsc{Decision}, baid, d)$.

Term 2 On receiving $(\textsc{TimeCap}, d)$ from $2t + 1$ parties, terminate.

9.4 Weak Multi-Valued Byzantine Agreement

- **Weak Multi-Valued Byzantine Agreement** allows one to use Byzantine Agreement for much larger decisions than 1 bit
  - It is called weak since it is not guaranteed to agree on an input of an honest party
  - If it decides on a value that was not a correct input, then it is a special error symbol $\perp$
- Each Byzantine pre-agreement will be identified by a fresh identifier *baid*
  - Each party can get an input $(\text{VOTE}, baid, v_i)$, where $v_i \in \{0,1\}^*$ is called the proposal
  - Each party can give an output $(\text{DECISION}, baid, d_i)$, where $d_i \in \{0,1\}^* \cup \{\top\}$ is called the decision
  - The properties required are

Agreement: If for some *baid* some correct P_i and P_j output $(\textsc{Decision}, baid, d_i)$ and $(\textsc{Decision}, baid, d_j)$, then $d_i = d_j$.
Weak Validity: If some correct P_i gives output $(\textsc{Decision}, baid, d)$ with $d \neq \perp$, then at least $n - 3t$ honest parties had input $(\textsc{Vote}, baid, d)$.
Preagreement Validity: If for some *baid* there exists d such that all correct P_i give input $(\textsc{Vote}, baid, d)$, then no correct P_i gives an output $(\textsc{Decision}, baid, d' \neq d)$,
Termination: If for some *baid* all correct processes got an input of the form $(\textsc{Vote}, baid, \cdot)$, then eventually all correct processes give an output of the form $(\textsc{Decision}, baid, \cdot)$.

- In the protocol there are n parties and a parameter $t < n/5$
  - There are at most t maliciously faulty parties
  - The protocol uses a BA protocol which is assumed secure for the parameters
  - The protocol:

1. On input (VOTE, $baid, d$), broadcast (VOTE, $baid, d$) to all parties using Byzantine broadcast.
2. Collect inputs (VOTE, $baid, d_i$) from $n - t$ parties. If there exists d such that $d_i = d$ for $n - 2t$ of the collected values, then let $v = 1$, otherwise let $v = 0$.
3. Run a BA with input v and wait for the output e.
4. If $e = 0$, then output (DECISION, $baid, \perp$). Otherwise, let d be the value sent by most processes in Step 2 and then output (DECISION, $baid, d$).

10 Key Management and Infrastructures (9)

10.1 General

- When solving the problem of distributing keys one has to be aware of the following problem: *Any secret system parameter runs a greater risk of being revealed, the longer time you keep it constant, and the more you use it*
  - It is good practise to change the keys one uses at regular intervals

10.2 Two-party Communication

- The standard way for achieving confidentiality (or authenticity or both) in two-party communication between A and B is to have a key K_{AB} agreed between A and B initially which is used for sending keys from A to B
  - To send a message M, A will generate a random `session` key k and send $E_{K_{AB}}(K), E_k(M)$ to be
    - K will be used for multiple messages but deleted after a short time e.g. when the connection is closed
  - The argument for using this system is to avoid given a potential adversary a very large amount of data encrypted under one key which makes the crypt-analysis easier
  - To solve the problem for multiple users one would need another solution since the number of keys needed to manage would grow quadratically with the number of users

10.3 Key Distribution Centers (KDC)

- **Key Distribution Centers** (KDC) are based only on secret-key technology
  - There are one KDC and many users
  - Every user A shares a key K_A with the KDC
  - When A wants to talk to B, the KDC generates a key K for the session and send $E_{K_A}(K)$ to A and $E_{K_B}(K)$ to B
    - Both parties can then recover K and communicate using this key
  - KDC solutions are not so common because of the single point of failure which the KDC represents because
    - All users must trust the KDC completely, since the KDC can decrypt or forge all traffic if it decides to misuse its knowledge.
    - If the KDC is down, the entire secure communication systems becomes unavailable.

10.4 Certification Authorities (CA)

- In the two player above the shared key K_{AB} could be replaced by a public key pair (sk_B, pk_B) generated by B
  - Only B knows sk_B and pk_B is public
  - A can send $E_{pk_B}(K)$ to B where K is the session key
    - This means no secret key needs to be shared initially but one still needs to ensure that A uses the correct public key
- Assume one have an entity called a Certification Authority with its own key pair (sk_{CA}, pk_{CA})
  - Assume that we can ensure that all users get an authentic copy of pk_{CA}
  - One can then do the following:

    * Each users A in the system must contact the CA in some way and send his public key pk_A to the CA
    * The way A identifies himself cannot be via cryptography only:
        · Before A has registered A and the CA do not share any secret keys
        · The CA does not know any public key it can safely assume belongs to A
    * There are many ways for A to identify himself
        · In rare cases A must show up in person
        · In other cases a pin code is sent to $\frac{1}{2}$A$ in paper mail and used to log into the website of the CA
    * If CA accepts the identity of the user it will issue a certificate which consists of
        · A string ID_A
        · The public key pk_A
        · The CA's signature $S_{sk_{CA}}(ID_A, pk_A)$ on the former two pieces of data
    * Once certificates are issued it is possible to set up a secure communication with necessarily involving the CA

– If a user suspects that his private key has been compromised, it should be possible for him to report this and have his certificate revoked
    * This means there must be an option to check whether a given certificate is still valid
    * The CA should implemented an on-line service for this purpose
    * The certificate should also contain a validy period as part of what is signed by the CA

– A certificate is typically a long data record with for instance the following fields:
    * Name of the certificate owner
    * Name of the CA who signed the certificate
    * Method used by CA to check identity of certificate owner

        * Date issued
        * Validity period
        * Rights and privileges of certificate owner.
        * Crypto algorithm to be used for checking this certificate
        * Crypto algorithm used by certificate owner
        * Public key of certificate owner
        * Signature of CA.
    - One can use different certificate authorities each others certificate public keys e.g. $Cert_{CA_1}(CA_2, pk_{CA_2})$
        * A more general way is to use certificate chains, which is an ordered list of certificates
            · Where the first entry is a certificate where CA_1 certifies the public key of user in question, in the second entry CA_2 certifies the public key of CA_1 etc. until the last entry where CA_n certifies the public key of CA_{n-1}
        * Limitations of certificate chain
            1. A should only trust the end point to the extent that he trust that every CA involved in the chain has not issued fake certificates
            2. Do not remove the need for at least one public key to be known to users initially
                · In a practical situation, the way that it is ensure is that the required public keys are delivered to the user together with the software needed to generate keys an do encryption and signatures

10.5 Limitations on Key Management

- *Any secure system using cryptography must make use of one or more keys that are protected only by physical, non-cryptographic means*

10.6 Password Security

10.6.1 General

- Passwords are often the weakest link in the security chain because they have to be remembered by humans

- There are (at least) 4 important security aspects of password security and therefore 4 types of attacks that one must protect against
  - How is the password chosen?
    - Can the adversary guess the password and verify his guess?
  - How is the password transmitted between the password user and verifier?
    - Can the adversary get hold of the password while it is in transit?
  - How is the password stored by the password user?
    - Can the adversary steal the password from the user?
  - How is the password stored by the password verifier?
    - Can the adversary steal the password from the verifier?

10.6.2 Choosing and guessing Passwords

- In general if the password is chosen from a character set of size C and has length ℓ then there are C^ℓ possible passwords
  - This number grows must faster as a function of ℓ than as a function of C
  - There are practical limitations to the length and studies show that 12 digits seem to be the maximum one can expect anyone to enter correctly
  - The length of the password is not a quality by itself, since passwords that real people can remember are usually not uniformly distributed
  - Some systems test chosen passwords against a know list of "bad" passwords and refuses people to use them
  - One should limited the number of failed login attempts and lock the account to make it harder for the adversary to guess passwords

        * This could create the a problem where the adversary makes the availability break down
        * A better idea is to have the system wait for some time after some failed login attempts before he is allowed to try again and make this time longer, the more failed attempts are maid

10.6.3 Using and eavesdropping passwords

- Stealing the password when it is used can take many forms
    - Such as looking over someones shoulder
    - Passwords sent over a LAN are very easy to detect and grab
        * e.g. hackers sitting at the local cafe intercepting other customers using the free wifi
    - Looking over someone's shoulder can also be done electronically using so called spyware where the code run on the victim's machine in some unnoticed way
        * The problem will record passwords and send them to the attacker
    - Encrypting network traffic definitely helps

10.6.4 Storing and stealing passwords, the user side

- Stealing the password can be easy if it is written down
    - There are other methods such as fool users into revealing their passwords under false pretenses
        * Known as **social engineering**
        * One of the most effective attacks on real systems and preventing it is very difficult
    - A variant of social engineering is a **phishing attack** where one sends a mail to the victim claiming that you are his bank or instance

    * The mails ask the user to following a link leading to a fake web page that claims to be his bank's
    * The user is then asked to type his user name and password, to "validate the account"

10.6.5 Storing and stealing passwords, the verifier side

- Stealing passwords from the verifying party may also be possible
  - In some bad cases systems store passwords in cleartext
  - Some systems such as UNIX store a user record containing username u and $f(pw_u)$ where
    * pw_u is the password of user u
    * f is a one-way function where it is easy to compute $f(pw_u)$ from pw_u but it is hard to go in the opposite direction
      · e.g. a hash function
  - If the attacker has the password file and a password pw that he thinks is the users passwords it is easy to verify by computing $f(pw)$ and check if it occurs in the file
  - *Password crackers* are programs that can crack some password files
    * They use dictionaries with probable password and try to match them with entries in a given password file
  - There are three main countermeasures against the password cracker kind of attacks:
    1. *Educate the users to choose hard to guess passwords*
    2. *Slow the attacker down:* It is possible to make the life of the adversary harder by making sure that the function f
       * is slow enough to slow down the number of attempts a password cracker can run per second
       * while being fast enough for the system to be efficient enough

3. *Removing the single point of failure:* an increasingly common architecture for stored password is for the password to be hashed together with a secret key K which is stored on a different machine than the one storing the password hash
    * The adversary must get hold of both the hashed password and the key K to be able to verify their guesses
    * Require the second server to be involved in every password verification attempt
    * When the user u register its password pw this is sent to the hashing server that computes $y = f(K, pw)$ and returns this value to the authentication server that stores the pair (u, y)
        · When the user contacts the authentication server, it sends the received password pw' and the hashed password y to the hashing server who checks whether $f(K, pw') = y$ or not

10.7 Hardware Security

10.7.1 Why use secure hardware?

- Hardware units with some degree of physical security are useful in many secure systems
    - The purpose is to prevent an adversary from getting hold of the secret keys
    - It is possible for an adversary to get a hold of a private signature key on a hard disk
        * The adversary then has lots of time offline to try and find a password by trying all possibilities and testing them against the encrypted key
        * Is also a problem with magnetic stripe cards
    - If the key is inside a hardware unit a potential attacker cannot easily break into the hardware unit
        * An adversary has to steal the hardware unit and needs time and money to break into the device

10.7.2 Tamper-evident hardware and two-factor authentication

- It is interesting to have hardware units that are difficult to break into even if it is not impossible to do so
  - It may still be useful in improving security as long as there is a significant cost, and if the attack will take some time and or may leave some trace on the device showing it was attacked
  - This type of hardware is often called **tamper-evident** hardware
  - Chip-cards are examples of such units
    - They often come with their own computer on board
      - The storage is physically protected such that access to using the keys on board should be only trough the cards own CPU
    - You must use the PIN code that opens the card and follow the communication protocol
    - They are not impossible to break into but it requires a skilled and determined attacker, it takes non-negligible time, and requires specialized equipment
  - The way these devices is programmed should be done carefully since it might reveal some information about the key
- A primary application for tamper-evident hardware is for two-factor authentication
  - The idea is that one authenticate a user in two ways: first by a password and second by checking that he has a certain hardware unit in his possession
  - Typically done by putting a secret key K inside the unit
    - This key is also held by the verifying party
    - The verifier issues a challenge c (a nonce) which the user forward to the hardware unit, it the returns a response $R(K, c)$ that is a function of both K and c
    - Typically it encrypts c under K

        * The response can clearly be verified by the other party and replay attacks will not work because the challenge will not be the same the next time
        * The response $R(K, c)$ is often called a **one-time password**
        * In some cases the challenge c is replaced by the current time which frees the user from manually forwarding c to the device
    - It can still be attacked using **real-time phishing** where the adversary fools the user to go to his web page rather than that of the bank and log in there instead
        * The adversary then logs into the bank at the same time

10.7.3 Tamper-Resistant hardware

- These are called **tamper resistant** or **tamper-proof** are much more difficult to break into than tamper evident hardware
    - It is estimated that not even a well funded organization with expert knowledge can break into the device
    - Some of them have their own CPU, storage and battery back-up and can do all the standard cryptographic algorithms internally.
    - Banks use these units to protect particularly sensitive and long-lived data, such as PIN codes for credit cards.
    - Also CA's use this kind of technology to store their private keys used for issuing

10.8 Biometrics

- **Biometrics** is using the various physical characteristics of the person trying to get access
    - It can take a number of different forms: it can scan your fingerprint, your eye, your face, or listen to your voice
    - The the major problem in this area is always to do the conversion to digital and the comparison such that
        * The system is tolerant enough to accept the good guys

        * The system is restrictive enough to reject the cheaters.
    - Biometrics can provide better access control for your private signature key
        * It cannot replace cryptographic authentication
    - Once has to take into account the weakest link such as a database containing the measurement

10.9 Preventing bypass of the system

- It is important that the system cannot be bypassword
    - I.e. it must not be possible to get to the resource without asking the system
    - If one has secure hardware available and the resource is a key this is not difficult
        * Just put the key inside the hardware and make sure it cannot be output from the device
    - If no secure hardware is available, life is much more difficult
        * This is the situation if one is designing security on a standard PC
        * If one want to protect a private key one can encrypt the key using the password as a key
            · To ensure that what we encrypt with is of fixed length the password is often hashed using some standard hash function h and using 128 bits of the output as key
            · For a password pw and secret key sk what is stored is $E_{h(pw)}(sk)$
        * Since the number of possible password is often must possible that the 2^{128} AES keysome precautions must be taken, so a potential attacker can't just try all possible possword
            · A solution is to make the function h slow i.e. rather than having h to be a standard hash function we make many iterations of it
            · This means that the right users will be slowed down, but not to much, but the attacker will have a much hard time

11 State Machine Replication (10)

11.1 General

- Replication is to run a service on several computers and keep them consistent
  - To give input to the system you give the input to at least one correct serve, which then will send it to the other servers
  - They all end up with an up-to-date copy of the service
  - There must be some mechanism in place which ensure that updates are applied in the same order at all replicates
  - If there is disagreements one needs to adopt the one that was sent by the most severs

11.2 State Machines

- The service is abstracted to be replicated by a state machine M
- **Definition 10.1 (State Machine)** A state machine M which consists of:
  - A set States
  - A start state $\text{State}_0 \in \text{States}$
  - A set Inputs
  - A set Outputs
  - A transition function $T : \text{States} \times \text{Inputs} \to \text{States} \times \text{Outputs}$
- A state machines starts in State_0
  - When it was in state State_i and receives input x then it computes $(State_{i+1}, y) = T(\text{State}_i, x)$, changes state to State_{i+1} and outputs y

11.3 Replicated State Machines

- A replicated state machine is a protocol for n servers which makes them behave as if they are running one single state machine M
- **Definition 10.2 (Replicated State Machine)** Let M be a state machine. A replicated state machine running M is specified via an ideal functionality RSM_M for n servers $\mathsf{S}_1, \ldots, \mathsf{S}_n$.
  - The syntax is as follows:
    - There is a protocol port IO_i for receiving inputs from server S_i and giving outputs to S_i
    - There is a special port RECEIVED_i for reporting what messages have been input to the ideal functionality by S_i
    - There is a special port PROCESS for specifying which messages to process next
    - There is a special port DELIVER_i for instructing the ideal functionality to deliver the next message to S_i
  - The ideal functionality runs as follows:
    - Upon initialization
      - Let $\text{State} = \text{State}_0$
      - For each IO_i, let Q_i be the empty queue
      - Q_i is for the outputs for server S_i which have not been delivered yet
      - Initialize UnProcessed to be the empty set
    - On input x on IO_i, output x on RECEIVED_i and add x to Unprocessed
    - On input x on \$PROCESS\$m where $x \in$ UnProcessed
      1. Let $(\text{State}', y) = T(\text{State}, x)$ and update $\text{State} = \text{State}'$
      2. Add t into all the queues q_i
      3. Remove x from UnProcessed
    - On an input on DELIVER_i where Q_i is not empty, remove the front element y from Q_i and output y on IO_i

- One can formulate a liveness property which says that if x is added to UnProcessed, then it will eventually be processed and all outputs will eventually be delivered
- An important safety property is that the outputs that the parties see is always the result of running from the initial state on some sequence of inputs that is the same for all parties.
  - Notice that the parties using RSM_M might not have seen the same number of these outputs.

11.4 Totally-Ordered Broadcast

- The main building block to implement state-machine replication is totally-ordered broadcast
- **Definition 10.3 (Totally-Ordered Broadcast)** We specify the behavior of totally-ordered via an ideal functionality TOB for n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$
  - The syntax is as follows

- There is a protocol port IO_i for receiving inputs from server P_i and giving outputs to P_i.
- There is a special port $\textsc{Received}_i$ for reporting what messages have been input to the ideal functionality by S_i.
- There is a special port Queue for specifying which message to queue next.
- There is a special port $\textsc{Deliver}_i$ for instructing the ideal functionality to deliver the next message to P_i.

- The ideal functionality runs as follows

- On initialisation, let $\mathsf{UnQueued} = \emptyset$. For each IO_i, let Q_i be the empty queue. The queue Q_i is the outputs for P_i which have not been delivered yet.
- On input x on $\textsc{in}_i$, output x on $\textsc{Received}_i$ and add x to $\mathsf{UnQueued}$.
- On input x on Queue, where $x \in \mathsf{UnQueued}$, enter x into all the queues Q_i, and then remove x from $\mathsf{UnQueued}$.
- On an input on $\textsc{Deliver}_i$ where Q_i is not empty, remove the front element x from Q_i and output x on IO_i.

- To implement RSM_M given TOB one simply broadcast all inputs to all servers which then run the machine M on the agreed sequence

- Everyone runs the same machine on the same sequence of inputs and therefore all servers will end up in the same state

- Replicated State Machine for $M = (\text{States}, \text{State}_0, \text{Inputs}, \text{Outputs})$ from Totally-Ordered Broadcast

  The protocol has n servers S_i and uses TOB.

  - Each S_i has a port $\text{RSM}_M.\text{IO}_i$ for receiving inputs and giving outputs to P_i.
  - Each S_i has a port connected to TOB.IO_i so it can give TOB inputs and outputs.

  The protocol runs as follows:

  - S_i: On initialization, initialise TOB and let $\mathsf{State} = \mathsf{State}_0$.
  - S_i: On input x on $\text{RSM}_M.\text{IO}_i$ input x on TOB.IO_i.
  - S_i: On output x on TOB.OUT_i, let $(\mathsf{State}', y) = T(\mathsf{State}, x)$, update $\mathsf{State} = \mathsf{State}'$ and output y on IO_i.

11.5 Crypto Currencies I

- A popular use of state machine replications is known as **cryptocurrencies**

  - All accounts are identified by a verification vk and the account owner has the corresponding signing key sk

    * To spend from the account, the transaction needs to be signed by the corresponding sk

    * Ensures that only the account holder can spend any money on the account

  - It is important to use totally-ordered broadcast to be able to agree on the order in which transactions are made

    * If there is an attempt to transfer more money from an account than it holds, the latest withdrawal has to be cancelled

    * To be able to do that all parties need to agree which transaction was the last one

  - The internal state of the machine is a list of pairs (vk, h_{vk}), where vk where vk is the verification key for the signature scheme and $h_{\text{vk}} \in \mathbb{R}_0$ is the holdings of the account

- The holdings of the initial accounts is application specific
    - It is assumed tat some accounts $(\text{vk}, h_{\text{vk}})$ are build into the machine when it is created
    - The initial accounts will typically belong to the people that built the system and the ones that invested in the system
- On input $(\text{TRANSFER}, a, \text{vk}_S, \text{vk}_R, \sigma)$, where $a \geq 0$ and

$$\text{Ver}_{\text{vk}_s}(\sigma, (\text{TRANSFER}, a, \text{vk}_R)) = \top \tag{19}$$

- the machine will do the following:
    - If $a < h_{vks}$ then ignore the command
    - Otherwise if $(\text{vk}_R, h_{\text{vk}_s})$ exists, then retrieve it otherwise initialize it to $(\text{vk}_R, h_{\text{vk}_R} = 0)$ and update $(\text{vk}_S, h_{\text{vk}_S})$ to $(\text{vk}_S, h_{\text{vk}_S} - a)$ and update $(\text{vk}_R, h_{\text{vk}_R})$ to $(\text{vk}_R, h_{\text{vk}_R} + a)$

11.6 Synchronous Implementation of Totally-Ordered Broadcast

- The following is a trivial synchronous implementation of TOP
    - All parties broadcast their messages which ensure that all parties eventually see all the same messages
        - Which might be in different order
    - To fix the order a leader is elected who gets to decide the order
    - The protocol blueprint is as follows:

1. When a new command x arrives at S_i, use unscheduled consensus broadcast (see Section 7.4) to broadcast x to all servers. Recall that in unscheduled broadcast a party can broadcast when it wants to and then the message eventually arrive at all parties.
2. All servers S_i keeps a set $\mathsf{UnQueued}_i$ of all messages received above. These might have been received in different order.
3. All servers S_i also keeps a set Queued_i. They will be moving messages from $\mathsf{UnQueued}_i$ to Queued_i. When they do so they make sure to enter messages into Queued_i in the same order.
4. There is a designated leader L, often called a $\mathsf{sequencer}$, who will help the servers keep the same order when they put messages into Queued_i. If L is corrupted it might harm liveness but not safety, specifically, if message are put into Queued_i then all servers do it in the same order. However, it might happen that no messages are put into Queued_i. Liveness is then guaranteed by leading the leader role go on round robin.

   They way that the leader orders messages is that it broadcast a so-called block, which is essentially just an ordered list of messages that have still not been processed. Everybody outputs them in the ordered that the leader has chosen. It is important that all parties get the next block or not. Therefore we use scheduled broadcast, where everybody is guaranteed to get an output at the same time.[1] The reason we can use scheduled broadcast is that the leader is supposed to send the next block at a well-defined time.

• Synchronous Totally-Ordered Broadcast

A protocol for n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. It uses

- An unscheduled consensus broadcast UCB (see Section **??**).
- A scheduled consensus broadcast, where all correct parties terminate in the same round SCB (see Section 7.5).

The protocol consists of two parts. The first part works using the following activation rules.

- P_i: On input x on TOB.IO_i input x on UCB.IO_i.
- P_i: On output x on UCB.IO_i add x to $\mathsf{UnQueued}_i$.

The second part works using the following activation rules:

1. Initially, let $\mathsf{UnQueued}_i$ and Queued_i be empty sets. Let $\mathsf{epoch} = 1$. The value epoch indicates who is the leader in a given epoch. The leader in epoch epoch is the P_i with $i = \mathsf{epoch} \pmod n$.
2. Do the following:
   - Leader P_i: Let $U_i = \mathsf{UnQueued}_i \setminus \mathsf{Queued}_i$, and input U_i on SCB.IN_i.
   - $\mathsf{P}_{j \neq i}$: Input P_i on SCB.IN_j to indicate that P_i is broadcasting.
3. When the so-called block U is received on SCB.OUT_j, do the following: remove U from $\mathsf{UnQueued}_i$, add U to Queued_i, and output on TOB.OUT_j the elements in U in some deterministic order, say lexicographically. Let $\mathsf{epoch} = \mathsf{epoch} + 1$. Go to Step 2

11.7 Asynchronous Implementation of Totally-Ordered Broadcast

- The system consists of two different parts
  - The first on is a flooding system, which works using the following rules
    - P_i : On input x on TOP.IO$_i$ input x on UCB.IO$_i$
    - P_i : On output x on UCB.IO$_i$ ad x to Received$_i$
  - The second part of the system is very different from the synchronous version
    - The reason is that one cannot risk waiting for a particular leader
    - In each epoch all parties prose the next block and then make sure to wait until some honest parties had their block distributed to at least $t + 1$ other honest parties.
      - We say that such block was seen by many honest
    - Each honest parties collects from $n - t$ parties all the block that these parties have seen
      - This means that all blocks which where seen by many honest parties will be collected by all honest parties
    - The final block will be a union of these blocks
    - To detect which block were seen by all honest parties and ensure agreement on this some Byzantine agreement is used

11.8 Group Change

11.8.1 General

- The two protocols given assumed that the set of servers is static and that some of the servers are corrupted and some are correct

- The protocol uses an asynchronous Byzantine agreement ABA (see Section **??**).
- Initially let $\mathsf{epoch} = 1$ and $\mathsf{Delivered}_i = \emptyset$ and $\mathsf{Received}_i = \emptyset$. Also for each $e \geq 1$ let

$$\mathsf{Core}_i^e = \mathsf{CoreCandidates}_i^e = \mathsf{SeenByManyHonest}_i^e = \mathsf{HasSeen}_i^e = \emptyset \ .$$

  The use of the variables are as follows:
  - $\mathsf{HasSeen}_i^e$: This is the set of parties from which P_i saw a block in epoch e.
  - $\mathsf{SeenByManyHonest}_i^e$: This is the set of parties P_j for which P_i knows that at least $n - t$ parties claim to have seen the block from P_j.
  - $\mathsf{CoreCandidates}_i^e$: At a sufficiently late point in the protocol $\mathsf{CoreCandidates}_i^e$ will be set to be the union of the values $\mathsf{HasSeen}_j^e$ from $n - t$ other parties P_j. The logic of the protocol will ensure that in $\mathsf{CoreCandidates}_i^e$ there will be at lot of parties that will have their block seen by all honest parties.
- Each P_i runs the below activation rules. All values are sent via UCB.
  1. As the first thing in the new epoch, let $U_i^{\mathsf{epoch}} = \mathsf{Received}_i \setminus \mathsf{Delivered}_i$ and send $(\textsc{Block}, \mathsf{epoch}, U_i^{\mathsf{epoch}})$.
  2. When having received $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ from P_j, where $U_j^{\mathsf{epoch}} \subseteq \mathsf{Received}_i$, add P_j to $\mathsf{HasSeen}_i^{\mathsf{epoch}}$, store U_j^{epoch}, and send $(\textsc{SawBlockFrom}, \mathsf{epoch}, \mathsf{P}_j)$ with $(\textsc{SawBlockFrom}, \mathsf{epoch})$ being the message identifier.
  3. When having received $(\textsc{SawBlockFrom}, \mathsf{epoch}, \mathsf{P}_k)$ from $n - t$ distinct parties add P_k to $\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$.
  4. When it first happens that $|\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}| = n - t$, send the set $\mathsf{HasSeen}_i^{\mathsf{epoch}}$ to all parties.
  5. We say that $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ has arrived at P_i when the value was received and for each $\mathsf{P}_k \in \mathsf{HasSeen}_j^{\mathsf{epoch}}$ the value $(\textsc{Block}, \mathsf{epoch}, U_k^{\mathsf{epoch}})$ was received. When $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ has arrived from $n - t$ parties P_j, let

$$\mathsf{CoreCandidates}_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j} \mathsf{HasSeen}_j^{\mathsf{epoch}} \ .$$

  6. When $\mathsf{CoreCandidates}_i^{\mathsf{epoch}} \neq \emptyset$, then for each P_k run a $\mathrm{ABA}_k^{\mathsf{epoch}}$ with input 1 if $\mathsf{P}_k \in \mathsf{CoreCandidates}_i^{\mathsf{epoch}}$ and input 0 otherwise. Here (epoch, k) is the identifier.
  7. When all $\mathrm{ABA}_1^{\mathsf{epoch}}, \ldots, \mathrm{ABA}_n^{\mathsf{epoch}}$ terminated, then for each P_k let v_k^{epoch} be the output of $\mathrm{ABA}_k^{\mathsf{epoch}}$. Let $\mathsf{Core}_i^{\mathsf{epoch}}$ be the set of P_j for which $v_j^{\mathsf{epoch}} = 1$, i.e.,

$$\mathsf{Core}_i^{\mathsf{epoch}} = \{\mathsf{P}_j \mid v_j^{\mathsf{epoch}} = 1\} \ .$$

  8. When $\mathsf{Core}_i^{\mathsf{epoch}} \neq \emptyset$ and $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ arrived for all $\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}$, then let

$$V_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}} U_j^{\mathsf{epoch}} \ ,$$

  and

$$\mathsf{Delivered}_i = \mathsf{Delivered}_i \cup V_i^{\mathsf{epoch}} \ ,$$

  and output the elements of V_i^{epoch} in some deterministic order. Then let $\mathsf{epoch} = \mathsf{epoch} + 1$ and repeat the above activation rules for the new epoch.

Figure 7: Asynchronous Totally-Ordered Broadcast

- In practice a more realistic setting is that when some server is detected to be corrupted e.g. crashed, then it will be removed from the system and a new server will be added
  - There might be other reasons to remove or add severs
  - Such changes are called group change
  - The challenge is to let all servers in the system agree on who is part of the system and to add and remove servers while the system is running

11.8.2 Corruption Detection

- It is assumed that there is a subsystem CorruptionDetection which allows to detect which servers are corrupted
  - It is a system for some servers $\mathsf{S}_1, \mathsf{S}_2, \ldots$
  - It has the following protocols ports

- A port $\textsc{Accuse}_i$ where a server S_i can accuse some other server S_j of being corrupted. The input is of the form $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$. This can be due to seeing messages from S_j that it should not send or it could be due to S_j being excessively slow (for instance, it did not send a message for a week and the system administrator does not pick up her telephone.) All other sub-system have access to accusing.
- A port $\textsc{Report}_i$ on which the system reports to S_i whether S_j has been detected to be corrupted. The output is of the form $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

- We make the following requirements

Eventual Agreement: If an honest server outputs $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ then all correct servers eventually output $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

Validity: If an honest server outputs $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ then at some earlier point in time an honest server input $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

No False Positives: If an honest server inputs $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$, then S_j is corrupt.

- The third property is a **contract property**
  - If you are the user of the system, then the systems promises to have all its liveness and safety properties as long as you fulfil all the contract properties.

- The moment you break a contract property, all bets are off.

- A way to implement this is by signing the (IsCorrupt, S_j, epoch) and send it along with the message

  - If a server sees $t + 1$ correctly signed (IsCorrupt, S_j, epoch) for the same P_j is outputs(IsCorrupt, S_j, epoch)

11.8.3 Eviction

- Eviction of corrupted servers proceeds as follows

  - On output (IsCorrupt, S_j, epoch) on CorruptionDetection.IO$_i$, send
    (IsCorrupt, S_j, epoch)
    on TOB.IO$_i$.
  - If $t+1$ servers sent (IsCorrupt, S_j, epoch) on TOB then ignore all messages from S_j in all protocols. Simply do as if S_j stopped sending messages. At the same time, stop sending any messages to S_j over point-to-point channels.

- The eviction procedure is secure in any system tolerating Byzantine errors

  - A server is only evicted if it is corrupted

11.8.4 Entry

- To be able to add a new server and run it, the only information it needs to run the new epoch is epoch, Received$_i$ and Delivered$_i$

  - This is called the entry information

  - To be able to participate it needs to get this information from somewhere

  - It can get epoch by asking the network

    * Some servers might be slow and report an old number or no number at all

    * Other servers might be malicious ad report the wrong number

        * It is safe to adopt a too low number but not advisable to pick a too high one, since one would then do reentry from an earlier point and run a bit longer to catch up

    - If there are at most t corrupted servers, the following will do:

- S_i: Broadcast $(\textsc{Entry}, \mathsf{S}_i)$ on the totally-ordered broadcast.
- When S_j sees $(\textsc{Entry}, \mathsf{S}_i)$, it makes a connection to S_i and sends epoch_j.
- S_i: Collect epoch_j from $n-t$ parties. Remove the t largest numbers and adopt the largest one among the remaining ones.

- The following will let S_i learn Delivered

- S_i: Broadcast $(\textsc{Entry}, \mathsf{S}_i, \mathsf{epoch})$.
- When S_j sees $(\textsc{Entry}, \mathsf{S}_i, \mathsf{epoch})$, it makes a connection to S_i and sends $\mathsf{Delivered}_j^{\mathsf{epoch}}$: the value that S_j had for $\mathsf{Delivered}$ at the end of epoch epoch.
- S_i: Collect $\mathsf{Delivered}_j^{\mathsf{epoch}}$ from $2t+1$ parties and adopt the value that was sent by at least $t+1$ servers.

- Since the server S_i knows epoch and the set Delivered at the end of the period

    1. It sets $\text{Delivered}_i = \text{Delivered}$
    2. Since it is perfectly possible for a server to have $\text{Received}_i = \text{Delivered}_i$ it can now set $\text{Received}_i = \text{Delivered}_i$
    3. It is now a fully functioning node again

- For at server S_i to enter the flooding network and receive all messages that were sent in epoch epoch the following is done

    1. Enter the flooding network to start receiving all new message
    2. Ask $2t+1$ peers to forward old messages and use majority to find the right ones
    3. It adds all old and new messages to Received and starts running from state $(\text{epoch}, \text{Delivered}_i, \text{Received}_i)$
    4. It is now a fully operational new server

12 Blockchains (11)

12.1 General

- A **blockchain** is an implementation of totally ordered broadcast and can be used for anything where a totally-ordered broadcast is useful.
- A **cryptocurrency** is a layer that can be put on top of any totally-ordered broadcast, not just the blockchain-based ones.

12.2 Synchrony

- A notion of global physical time t is assumed
  - Each party P_i has a local clock Clock_i
  - A bound of MaxDrift is assumed on clock drift
    - It is assumed that $|t - \text{Clock}_i| \leq \text{MaxDrift}/2$ for all correct P_i
    - For two correct P_i and P_j one will have the relation $|\text{Clock}_i - \text{Clock}_j| \leq \text{MaxDrift}$
    - It could e.g. be done by each client synchronizing against a server

12.3 Flooding System

- It is assumed that there is a flooding system
  - A model is considered where parties can come and go
  - If a party is awake when a message is sent and stays awake for long enough then it is guaranteed to get the message
  - There is some fixed upper bound τ on the delivery time
- The ideal functionality is as follows:

- For each party P_i there is a protocol port FLOOD.IO$_i$ for sending and receiving messages. Each P_i also has a port PARTICIPATE.WAKEUP$_i$ and a port PARTICIPATE.GOTOSLEEP$_i$. There is also a special port TAU. There is a set Awake, which is initially empty. There is a time (seconds) parameter τ initially set to an arbitrary constant, say 42.
- If $\tau' > 0$ is input on TAU and this is the first time there is an input on TAU and no messages have been sent yet, then update $\tau \leftarrow \tau'$. This allows the environment to set some fixed upper bound on message delivery.
- The ideal functionality keeps a set Awake. On any input on WAKEUP$_i$ it adds i to Awake. On any input on GOTOSLEEP$_i$ it removes i to Awake.
- On input x on IO$_i$ where $i \in$ Awake, add x to InTransit$_j$ for all P_j.

- On input x on Deliver$_i$ where x is in InTransit$_j$, output x on FLOOD.IO$_i$.

- For a given message x
  - Let I_x be the time t when x was input to a correct P_i the first time
  - Let O_x be t when x was delivered at the last correct P_i that was in Awake since time I_c
  - It is assumed that $O_x - I_x > \tau$ never happens
  - In the case where x was input to a incorrect P_i the guarantee is as follows
    - If any correct outputs x at time I_x and P_i and P_j remains correct and alive to τ seconds, then P_j will also output x

12.4 Lottery System

12.4.1 General

- A lottery system LOTTERY is assumed
  - It breaks time up into slots of length SlotLength
  - A party P_i is in slot slot if

$$\text{slot} - 1 \leq \frac{\text{Clock}_i}{\text{SlotLength}} < \text{slot} \tag{20}$$

- There is a lottery system which allows a party to get a draw $\text{Draw}_{i,\text{slot}}$ in each slot

- Each draw has an associated value $\text{Val}(\text{Draw}_{i,\text{slot}})$
- The winner of the slot is the party with highest $\text{Val}(\text{Draw}_{i,\text{slot}})$
- The lottery proceeds as follows:

- Each party has a associated value $\mathsf{Tickets}_i$. Think of this as how many tickets P_i has bought. These numbers are know by all parties.
- In the first round each party generates a key pair $(\mathsf{vk}_i, \mathsf{sk}_i)$ for a signature scheme and vk_i is broadcast to all other parties. This signature scheme should have unique signatures such that for each vk_i and each message m there is at most one value σ such that $\mathsf{Ver}_{\mathsf{vk}_i}(\sigma, m) = \top$.
- In the second round a random number Seed is made public. It should be unpredictable by all parties in round 1.
- For each $\mathsf{slot} = 0, 1, \ldots$, party P_i can compute the draw

$$\mathsf{Draw}_{\mathsf{slot},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}) \ .$$

The value of a draw is defined as follows: If $\mathsf{Ver}_{\mathsf{vk}_i}(\mathsf{Draw}, (\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot})) = \bot$, then $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = -\infty$. Otherwise,

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = \mathsf{Tickets}_i \cdot H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw}) \ .$$

- Since all parties know all the values Tickets_i and vk_i, so all parties can compute $\text{Val}(P_i, \text{slot}, \text{Draw}$ for all $P_i, \text{slot}, \text{Draw}$
  - The reason unique signature is needed is that if P_i could compute several valid signature for $(\text{LOTTERY}, \text{Seed}, \text{slot})$, then a malicious server would get multiple attempts at winning the lottery
    - We would like a corrupt process to lose the lottery as often as possible
  - The function H will be a cryptographic hash function with 256-bit output
    - e.g SHA256
  - For analysis it is assumed that there is a random oracle that will output uniformly random values
    - Except that in the same input it always outputs the same value
    - These bit strings as thought of as a number between 0 and $2^{256} - 1$

– Since the outputs of H are assumed to be uniformly random in $[0, 2^{256})$ the probability that two draws will ever have the same value is negligible

  * Each slot has a unique winner except with negligible probability

  * To be completely sure we have a unique winner, let us that if two parties P_i and P_j have $\mathrm{Val}(P_i, \mathrm{slot}, \mathrm{Draw}_i) = \mathrm{Val}(P_i, \mathrm{slot}, \mathrm{Draw}_j)$ then we simply say that $\mathrm{Val}(P_i, \mathrm{slot}, \mathrm{Draw}_i) < \mathrm{Val}(P_i, \mathrm{slot}, \mathrm{Draw}_j)$ when $\mathrm{vk}_i < \mathrm{vk}_j$ in the lexicographic ordering

  * The short hand

$$\mathrm{Draw}_{\mathrm{slot},i} \leftarrow \mathrm{Draw}(\mathrm{sk}_i, \mathrm{slot} \tag{21}$$

is used for P_i computing its draw in the slot slot

12.4.2 A Protocol that Almost Works

- Assume that we have a flooding system and a lottery system

  – We first consider a protocol where all parties participate all the time

- P_i : Let slot_i be the slot number. Initially, let $\mathsf{Delivered}_i = \emptyset$. Let $\mathsf{Received}_i$ be the messages received by the flooding system.
- Each P_i runs the below activation rules. All values are sent via FLOOD.
  1. As the first thing in the new slot, get the draw Draw_i, let $U_i^{\mathsf{slot}} = \mathsf{Received}_i \setminus \mathsf{Delivered}_i$ and send $V_i = (\textsc{Block}, \mathsf{slot}, \mathsf{Draw}_i, U_i^{\mathsf{slot}})$ along with $\sigma_i = \mathsf{Sig}_{\mathsf{sk}_i}(V_i)$. Let $\mathsf{cw} = i$ and store $(\mathsf{slot}, \mathsf{P}_{cw}, \mathsf{Draw}_{\mathsf{cw}}, U_{\mathsf{cw}}^{\mathsf{slot}})$ as the current winner of this slot.
  2. Until the end of slot slot collect incoming values $(V_j = (\textsc{Block}, \mathsf{slot}, \mathsf{Draw}_j, U_j^{\mathsf{slot}}), \sigma_j)$. If $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}_j) > \mathsf{Val}(\mathsf{P}_{\mathsf{cw}}, \mathsf{slot}, \mathsf{Draw}_{\mathsf{cw}})$, then store $(\mathsf{slot}, \mathsf{P}_j, \mathsf{Draw}_j, U_j^{\mathsf{slot}})$ as the current winner and let $\mathsf{cw} = j$.
  3. At the end of slot slot, let $(\mathsf{slot}, \mathsf{P}_{cw}, \mathsf{Draw}_{\mathsf{cw}}, U_{\mathsf{cw}}^{\mathsf{slot}})$ be the current winner. Output $U_{\mathsf{cw}}^{\mathsf{slot}}$ on TOB.IO$_i$ in some deterministic order and let $\mathsf{Delivered}_i = \mathsf{Delivered}_i \cup U_{\mathsf{cw}}^{\mathsf{slot}}$.

- **Lemma 11.1** Let MaxDrift be the maximal clock drift in the network and let τ be an upper bound on the time it takes to deliver a message. If SlotLength > MaxDrift + τ then the following holds

  – If $\mathrm{P}_{\mathrm{win}}$ is honest in slot slot, then all correct P_i will have $\mathrm{cw}_i = win$ at the end of slot slot

  - If all parties agreed on Delivered$_i$ at the beginning of slot \$slot4, then all correct parties agree on Delivered$_i$ at the end of slot slot

- When P_{win} is correct, then Delivered grows with all the values that reached P_{win} before the beginning of slot slot

  - The system is live in the sense that it does not take a message x longer to be delivered than it takes to reach all parties and then a correct process to win

12.4.3 The Problem

- The problem with the protocol happens when P_{win} is corrupted

  - A corrupted P_{win} can instead of sending V_{win} at the beginning of slot slot it waits until for instance $\text{t}_{\text{win}} + \tau/2$

    * i.e. it sends it about halfway into the slot

  - As a result some parties might receive V_{win} and set $\text{cw}_i = \text{win}$ and some might not have time to receive V_{win} and therefor set $\text{cw}_i = \text{rup}$ where P_i is the runner up

    * i.e. the parties with the second highest values is draw in slot slot

    * agreement is lost

12.4.4 Creating more Problems

- The problematic situation discussed happens where the slot winner is corrupted and sends the block late, then some correct parties might have different $B_i = \text{BestLeft}(\text{Tree}_i)$ at the end of the slot

- It is important for the protocol that $\text{SlotLength} > \text{MaxDrift} + \tau$

  - It turns out that the new protocol can tolerate MaxDrift+MaxDeliveryTime > SlotLength if pnly it does not happen to often

  - It is important for efficiency as there will some variation in drift

        * If one has to set MaxDeliveryTime such that it holds except with probability 2^{80} that the delivery time is never less than MaxDeliveryTime throughout the lifetime of the system
        * If on the other hand we have to set it such that with probability 95% it holds then the delivery time can be set to much lower

- An important optimization that allows to save a lot of bandwidth
    - In the above protocol each party sends a block to each other party in each slot which creates a lot of traffic
    - A hardness threshold Hardness is introduced
    - Only parties with a ticket with value higher than Hardness will send its block
    - If we set Hardness high enough this will ensure that only a few blocks a sent to all parties in each slot
        * Gives a dramatic optimization in bandwidth
    - As a consequence it can happen in some round that not block is send because all ticket has a value less than Hardness

12.5 Growing a Tree

12.5.1 Protocol

- There are two types of problems to solve
    1. Sometimes a winning block might get delivered only to some correct parties
        - Either because a block is sent late with malice or because the delivery time is too high in the lot
    2. Sometimes there is no slot winner
- The solution to both of these problem is to grow a tree instead of a chain

- Each party is supposed to take the longest path in the tree as its chain
- This will tend to converge on agreement

- **Definition 11.2 (block tree)** A node is of the form $(\text{BLOCK}, \text{P}_j, \text{slot}, \text{Draw}, U, h, \sigma)$
  - They mean the following
    - BLOCK just specifies the type of the tuple
      - That it is a block
    - slot $\in \mathbb{N}$ is a block number
    - Draw is the draw that was used to win the lottery
    - U is the block data
    - h is a block hash (of some previous block)
  - The value of a block N is defined to be $\text{Val}(N.\text{P}, N.\text{slot}, N.\text{Draw})$
  - There is a special block $(\text{BLOCK}, \perp, 0, \perp, U_0, \perp)$ with value ∞ called the genesis block, where U_0 is the genesis data
    - It contains Hadness and more
    - It is sometimes just called G
  - A set S of blocks which contains G and where all blocks are valid defines a tree as follows
    - The nodes of the tree is a subset of the nodes it Tree
    - The root of the block tree is the genesis block G
    - Edges are directed and points towards the root
    - There is an edge to $N_1 \in$ Tree from $N_2 \in$ Tree if and if $N_2.h = H(N_1)$ and $N_2.\text{slot} > N_1.\text{slot}$
    - For a given node N we let $\text{ParthTo}(N)$ be the list of nodes from G to N including G and N and indexet from 0

- An important component in block chains in the notion that some path in the tree being better than other
  - In general we just like the longest path the best
    - This is the one we want to build on.
  - If two paths have the same length a tie breaker might want to be used
- **Definition 11.3 (path weight)** A blockchain is a block tree where each block has at most one parent
  - A path weight is a function PathWeight mapping blockchains in a total ordered set
  - It should have the following properties
    - If $P^{'}$ is a proper prefix of P, then PathWeight($P^{'}$) < PathWeight(P)
    - If $P \neq P^{'}$ then PathWeight($P^{'}$) $\neq$ PathWeight(P)
    - It cannot happen that PathWeight($P^{'}$) < PathWeight(P) and Len(PathWeight($P^{'}$)) > Len(PathWeight(P))
  - The default path weight is just the order which sort first on length of P and the Val(Left(P))
    - So PathWeight(N_1) < PathWeight(N_2) if and only if Len(N_1) < Len(N_2) or Len(N_1) = Len(N_2) and Val(N_1) < Val(N_2)
- **Definition 11.4 (best path, best leaf)** Let Tree = BlockTree(S) be a block tree then the best path in Tree in the path from G to a left that maximizes PathWeight
  - We write P = BestPath(Tree)
  - The best left of Tree is BestLeaf(Tree) = Leaf(BestPath(Tree))
- The protocol uses two auxiliary functions
  1. `GetMetaData` will ask the local system is there extra data to include in the block data

– Assume that GetMetaData$(\cdot) = \emptyset$ for now

2. `ValidMetaData` will check wheter some meta data is valid

– Assume that ValidMetaData$(\cdot) = \top$

- The protocol proceeds as follows

- P_i : Throughout the protocol let slot_i denote the current slot number. Initially $\mathsf{slot}_i =$. Initially, let $\mathsf{Delivered}_i = \emptyset$. Let $\mathsf{Received}_i$ be the messages received by the flooding system. Let G be the genesis block and initially let $S_i = \{G\}$. Throughout the protocol let $\mathsf{Tree}_i = \mathsf{Tree}(S_i)$.
- Each P_i runs the below activation rules. All values are sent via $\textsc{Flood}$.

1. Immediately when slot slot_i is entered, let $P = \text{BestPath}(\mathsf{Tree}_i)$, get the draw Draw_i. If $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}) \geq \mathsf{Hardness}$, then proceed as follows. Let $M_i =$ GetMetaData, let $U_i = \mathsf{Received}_i \setminus \mathsf{Delivered}_i(P)$, let $B_i = \text{Leaf}(P)$, let $h_i = H(B_i)$, let $\sigma_i = \mathsf{Sig}_{\mathsf{sk}_i}(V_i)$, and send $V_i = (\textsc{Block}, \mathsf{P}_i, \mathsf{slot}_i, \mathsf{Draw}_i, (U_i, M_i), h_i, \sigma_i)$.
2. On input $V_j = (\textsc{Block}, \mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}, (U, M), B, \sigma)$ with $\mathsf{Ver}_{\mathsf{vk}_j}(\sigma_j, V_j) = \top$ and $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}) \geq \mathsf{Hardness}$, store it and wait until it happens that $\mathsf{slot}_i > \mathsf{slot}$ and ValidMetaData$(M) = \top$, then add it to S_i.

12.5.2 How does the tree grow

- **Definition 11.5 (honest tree)** Given two valid trees Tree_i and Tree_j the union $\text{Tree}_i \cup \text{Tree}_j$ is simply the tree with root G that contains all the paths of both Tree_i and Tree_j

  – This is again a valid tree

  – Let H be the set of all correct P_i at time t and let Tree_t be the tree of P_i at this time

  – Let

$$\text{HonestTree}^t_{\text{slot}} = \bigcup_{i \in H} \text{Tree}^t_i \tag{22}$$

- By definition all correct parties at any time sees a tree which is a subset of the honest tree

  – For all correct P_i clearly $\text{HonestTree}^t = \text{HonestTree}^t \cup \text{Tree}^t_i$

  – Furthermore if δ_t is the network delivery at time t and P_i is a party which is alive at time t until time $t + \Delta_t$ then

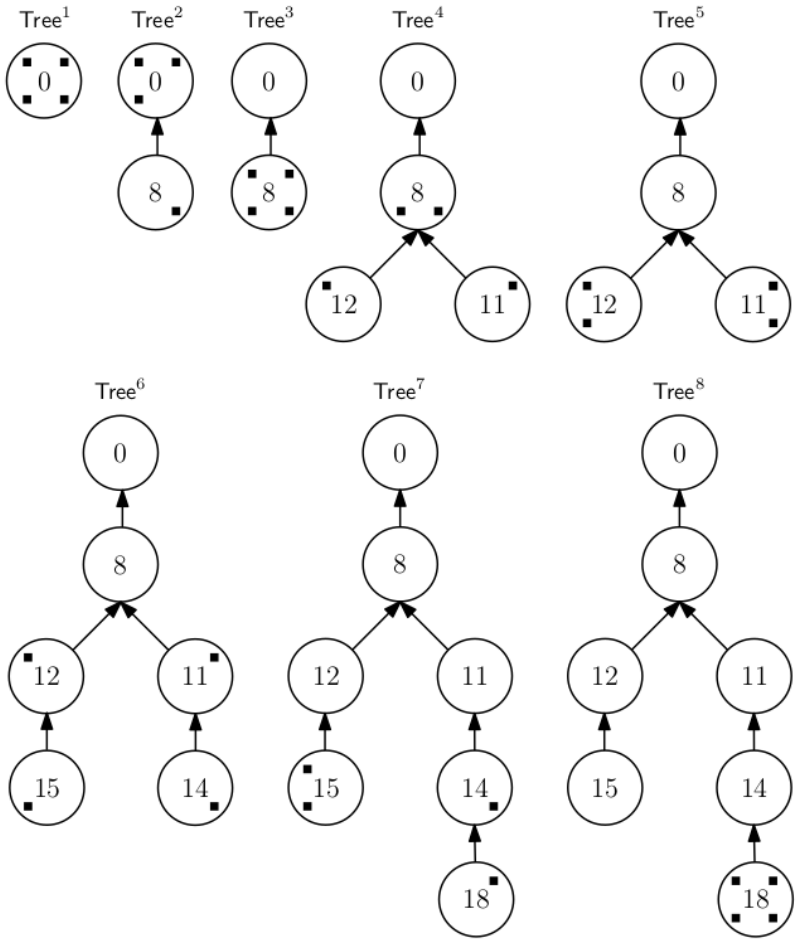


Figure 8: Example of tree growth

$$\text{HonestTree}^t \subseteq \text{Tree}^{t+\delta_t} \tag{23}$$

- as all nodes seen by any honest party at time t will reach P_i by time $t + \Delta_t$
  - If for long enough the honest tree does not grow then all honest parties will in fact see the honest tree and therefore agree on the best leaf
  - So for long enough between slot winners, all honest parties tend to build on the best path which will therefore get better and better than all small unlucky forks

12.5.3 Rollback and Finalization

- A big problem with tree protocols is when to deliver a transaction
  - When a party changes from a chain to another chain it is called **rollback**
  - If we want to avoid rollbacks, parties cannot deliver on IO_i any transaction that is in a branch that might later be rolled back
  - There are two different approaches to deciding when that has happened.
    1. A way is to just wait "long enough" and only consider a block safe when it is long enough up in the tree
    2. Another approach is to run a separate process which detects which blocks are final, called a finalization layer

12.5.4 Ghost Growth

- To understand how far one needs to look up in the tree to find a block that will never be rolled back we need to much better how the tree grows
  - Since winning a slot just means that $\text{Val}(P_i), \text{slot}, \text{Draw}) \geq \text{Hardness}$ then if a corrupt party wins slot slot it can for all U and all previous block B with a small slot number produce a block $(\text{BLOCK}, P_k, \text{slot}, \text{Draw}, U, H(B), \sigma)$ which will be valid

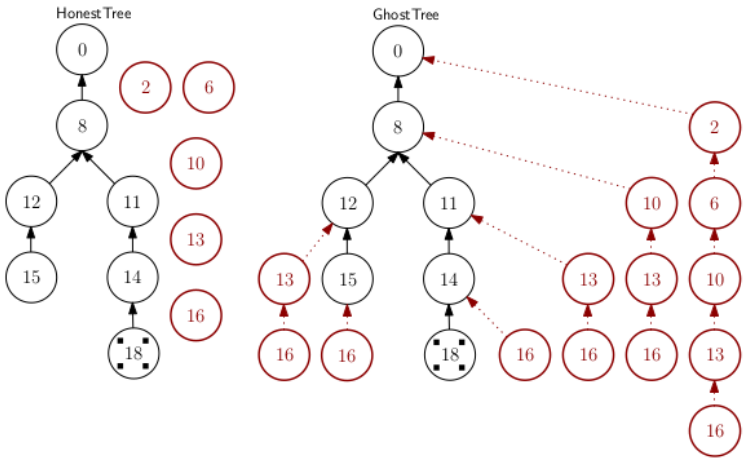


Figure 9: Example of a GhostTree

- A **ghost tree** is a tree that is obtained by starting with the honest tree and for each withheld winning draw, and add the corresponding blocks at all possible places
    - In particular the corrupted parties can add a new block at any node in the ghost tree with a smaller slot number
- The GhostTree at time t is denoted by GhostTree^t
- By creating long ghost trees one can get long rollbacks
- One solution which often comes to mind for limiting the problem
    - This is a bad idea since this might result in some parties rolling back and another which do not

12.6 Understanding Tree Growth

12.6.1 General

- In understanding tree growth, three properties have crystallized out of the current scientific literature called
    1. Tree growth

2. Chain quality

3. Limited rollback.

12.6.2 Tree Growth

- Tree growth basically says that the tree get higher all the time as there are more and more slot winners
  - It is parameterized by a fraction TreeGrowth which says that about a fraction Treegrowth of the slot winner makes the tree get higher
- **Definition 11.6 (tree growth)** Let TreeGrowth $\in [0, 1]$ be a real number
  - We say that a protocol has a **tree growth** of TreeGrowth if after n slot winners the height of HonestTree is at least TreeGrowth $\cdot$ $(n - \kappa)$ except with negligible probability $2^{-\kappa}$
- **Definition 11.7 (timely slot)** It is defined what it means for a slot slot to be timely
  - Let t be the time the slot starts
  - We say slot slot is timely if all messages before time $t-$MaxDeliveryTime are delivered before time t
- We call slot slot a **honest slot** if there is at least one honest winner in slot slot
  - It might have corrupted winners too and several honest winners
  - A slot slot is called a lucky slot if it is a timely slot and a honest slot
- **Lemma 11.8 (lucky slot)** Consider a timely slot slot and let HonestTree^t be the honest tree when the slot begins
  - Assume that there is an honest winner in slot slot
  - Let HonestTree$^{t-\text{MaxDeliveryTime}}$ be the honest tree MaxDeliveryTime seconds before. Then

$$\text{Len}(\text{BestPath}(\text{HonestTree}^t)) \geq \text{Len}(\text{BestPath}(\text{HonestTree}^{t-\text{MaxDeliveryTime}}))+1 \tag{24}$$

- **Definition 11.9 (wasted honest winners)** We call an honest slot winner wasted if it wins a slot which is not timely or it wins a timely slot which has another honest winner with a value of its
  - We let $\text{WastedHonestWinners}^t$ be the number of wasted honest slot winner at time t
- **Theorem 11.10 tree growth** Let HonestWinners^t be the number of honest winner at time t. Then

$$\text{Len}(\text{BestPath}(\text{HonestTree}^t)) \geq \text{HonestWinners}^t - \text{WastedHonestWinners}^t \tag{25}$$

- There will be about 59% lucky slot among the slots with a winner
  - Means that the tree grows is 59%
  - There will tend to be about 34% corrupt slots

12.6.3 Chain Quality

- Chain quality asks that a lot of the nodes on the best path were produced by honest parties
  - In the extreme case where all nodes are produced by corrupted parties, they might censor some transactions by not adding them to their blocks
  - If only few nodes are produced by honest parties, the throughput of censored transactions can be lowered by the corrupted parties which can be bad enough
- **Definition 11.11 (chain quality)** Let $\text{ChainQuality} \in [0, 1]$ be a real number
  - We say that a protocol has a chain quality of ChainQuality if after n slot winners the best path in the best path in HonestTree has $\text{ChainQuality} \cdot L - \kappa$ nodes that were produced by parties which were honest when they produced the node, except with negligible probability $2^{-\kappa}$
  - The protocol in the standard tree scenario has chain quality about 42%

12.6.4 Ghost Eventually Die

- Ghost chain that are kept hidden for too long will eventually become much shorter than the honest tree and therefore cannot be used for a rollback
    - Any ghost branch will eventually fall behind by κ nodes except with negligible probability

12.6.5 Limited Rollback

- The limited rollback property ask that there is a limit RollbackLimit such that the probability that there ever are a rollback longer than this is some negligibly small probability
- **Super slot** is defined to be one which was won by a honest party where the slot was timely and no other party either honest nor corrupt party won the slot
- **Lemma 11.12 (super slot)** All super blocks it at different heights in the honest tree
- All other blocks which are not super blocks are called **filler blocks**
- **Lemma 11.13 (super slot)** Assume that a rollback is possible from branch B_1 to branch B_2 both rooted at node N. If since N was created there were created s super blocks, then at least $s/2$ distinct filler blocks were created
- If one want treasonable security one should never expect to get lower than tolerating rollbacks of length 40

12.7 How to Buy Tickets

12.7.1 General

- One approach for buying tickets that does not work is to let people make accounts as they desire and say each account has 1 tickets
    - If there are 100 honest parties running the protocol and one corrupted party, the corrupted party could simply create 1000 fake accounts ad take over the network
    - Know as a Sybil attack

12.7.2 Permission Blockchains

- An easy solution to the buying ticks problem is to verify peoples identity when they create an account
  - This can e.g. be done if ten big companies run a blockchain between them: they simply get to have one account each
  - The assumption on having a majority of honest tickets is then translated into an assumption that a majority of the companies are honest.

12.7.3 Proof of Stake

- In a fully open peer-to-peer system one cannot control who has an account in the system
  - Anyone should be able to enter the system
- A popular way to buying tickets when the totally-ordered broadcast is used to implement a cryptocurrency is to make the tickets proportional to the holdings of ones account
  - The assumption is that the owner with the most money in the system are honest
  - Holders of money would have an interest in keeping the system healthy

12.7.4 Proof of Work

- A way to fight Sybil attacks is to use **proofs of work**
  - Sometimes called one clock-cycle, one ticket
  - Since you cannot cheat with how many clock cycles you have, they cannot be copied or freely created
    - Build on the principles of one watt, one ticket
- In a proof-of-work systems you typically do not win the right to fill the next slot, instead you win the right to extend the tree at a particular node

- To replace the Ticket_i's place with something proportional to how much computing power one put the solution to a hard computation puzzle
    - The harder a puzzle you can solve the more computational power you must have

- A example of a puzzle is given a hash function H, which maps into e.g. 256 bit strings find a number d such that the first h bits of $H(d)$ is 0
    - Since H generated random looking string the probability that the first h bits of $H(c)$ are all 0 is 2^{-h}
    - It can be proven that if you have an experiment which is successful with probability T^{-1} each time you repeat the times one has to repeat it is T times
    - The expected number of times one has to compute H is 2^h before finding the d
        - Therefore presenting a number d such that $H(d)$ has h trailing 0s is reasonable proof that you have computational power
    - It does not completely work, since a malicious party could copy an honest d
        - To prove that you did the work, one will hash along the identity which in this case is the verification key vk_i corresponding to your secret key sk_i
        - To prove your computational power one has to find d $H(\text{vk}_i, d)$ such that it starts with h zeroes
    - The value for ones draw is

$$\text{Val}(\text{P}_i, \text{slot}, \text{draw}) = (2^{256}/H(\text{vk}_i, d)) \cdot H(\text{LOTTERY}, \text{Seed}, \text{P}_i, \text{slot}, \text{Draw}) \tag{26}$$

- Since the number 2^{256} is the same for all parties one might as well set

$$\text{Val}(\text{P}_i, \text{slot}, \text{draw}) = H(\text{LOTTERY}, \text{Seed}, \text{P}_i, \text{slot}, \text{Draw})/H(\text{vk}_i, d) \tag{27}$$

- *Proof-of-Burning-the-Planet*: A strategy for finding $H(\text{vk}_i, d)$ is to enter the network and start computing $H(\text{vk}_i, c)$ for all c's and continuously broadcasting the smallest one
  - Would give a proof of total amount of energy spent
- *Proof-of-Hardware-Cost*: One could somehow broadcast a new seed Seed every week and then have all parties compute $H(\text{Seed}, \text{vk}_i, c), H(\text{Seed}, \text{vk}_i, c+1) \ldots$ for ten minutes and then broadcast the smallest $H(\text{Seed}, \text{vk}_i, d)$ that they found
  - By having Seed be random and unknown until the competition starts we could ensure that no starts before
  - The machinery for mining would only be turned on for ten minutes every week
  - It would burn magnitude less energy
- *Towards Bitcoin*: Bitcoin uses the following Val for the next slot

$$\text{Val} = H(\text{LOTTERY}, \text{Block}, \text{vk}_i, d)^{-1} \tag{28}$$

- The block Block is what the miner considers as the best leaf
  - The hash that won the lottery is added to Block in the block header
    - The block can then at the same time function as the next seed
    - Since you are certain of a "random" output of a hash function
  - Mining goes on constantly and therefore Bitcoin is basically a *Proof-of-Burning-the-Planet* system

12.8 Finality Layers: Pruning Ghost Branches

12.8.1 General

- There will be a finalization committee FinalityCommittee, which is a set of verification keys vk_i
  - Could e.g. consists of servers that we have seen are able to stay online stably for a long time
  - Members of the committee are called the finalisers
  - Each vk_i has a number of votes, denoted by Votes_i
    - Could e.g. be the sum of tickets of parties from the tree layer that delegated their voting power to the delegate or their number of tickets in the tree layer
- It is important that the finalization committee is mostly honest
  - It is assumed that at most one-third of the votes are corrupted
  - For a set S of finalisers let

$$\text{Votes}_s = \sum_{i \in S} \text{Votes}_i \tag{29}$$

- Let Corrupt be the set of the corrupted finalisers It is required that

$$\text{Votes}_{\text{Corrupt}} < \text{Votes}_{\text{FinalityCommittee}}/5 \tag{30}$$

- It is assumed that we have a weak multi valued BA WMVBA
  - It should be secure as long as $\text{Votes}_{\text{Corrupt}} < \text{Votes}_{\text{FinalityCommittee}}/5$
  - The following is the protocol AGREEONBLOCK(fid, c, h, Δ) that agrees on the block at height h

1. It is required that Δ is a integer and that $\Delta \geq 1$.
2. Wait until $\text{Len}(\text{BestPath}(\mathsf{Tree})) \geq L + \Delta$.
3. Let $\mathsf{Block}_i = \text{BestPath}(\mathsf{Tree})[h]$.
4. Let $baid = (fid, \Delta)$ and run $\mathsf{Block} = \text{WMVBA}(baid, \text{BLOCK}_i)$.
5. If $\mathsf{Block} = \perp$, then let $\Delta = 2\Delta$ and go to Step 2.
6. If $\mathsf{Block} \neq \perp$, then wait until Block occurs in the tree at height h in some branch and then output (Block, Δ).

- Input fid is a unique identifier for this run of the protocol
- Input c is a counter of previous finalization's
- Input h is the height of the block to finalize
- Input Δ is a "delay" in when to start the finalization
- The following is the protocol KEEPAGREEING(fid) for continuously agree on a higher and higher block:

1. Let $\Delta = 1$. Let $h' = 0$. Let $h = 1$. Let Agreed denote the height and value of the latest agreed block. Initially $\mathsf{Agreed} = (0, \mathsf{Block}_0)$, where Block_0 is the genesis block. Let $\mathsf{Agreeds} = \{\mathsf{Agreed}\}$.
2. Wait until $\text{BestLeaf}(\mathsf{Tree}).\mathsf{LastAgreed} = h'$.
3. Run $(\mathsf{Block}_h, \Delta_h) = \textsc{AgreeOnBlock}(fid, h', h, \Delta)$.
4. Let $\mathsf{Agreed} = (h, \mathsf{Block}_h)$. Let $\mathsf{Agreeds} = \mathsf{Agreeds} \cup \{\mathsf{Agreed}\}$.
5. Let $h' = h$. Let $h = h + \Delta_h$.
6. If $\Delta_h > 2$, then let $\Delta = \Delta_h/2$. Otherwise let $\Delta = 1$.
7. Let $c = c + 1$ and go to Step 2.

12.8.2 How do We Make a Final Block Final?

- To make sure there are no rollbacks from the blocks in Agreed the PathWeight rule is updated in such a way that makes sure they are part of the current best block
- **Finalization Meta-Data:** When a new block Block is made the meta data M is included as a new field LastAgreed
  - It is the maximal in the path from genesis to Block where a block from Agreeds is sitting
  - This is done via `GetMetaData()`
  - When a block Block is received it is not considered valid until PathTo(Block[Block.M.lastAgreed] $\in$ Agreeds
    - The Block.M.lastAgreed should be larger that LastAgreed on all nodes in PathTo(Block)
    - This check is done by ValidMetaData
- **Prefer-Agreed-Blocks Rule**: We update the PathWeight rule such that the number of finalized blocks counts first

- For any PathWeight let PathWeight$^{\text{FIN}}$ be the ordering which first sorts on Block.M.LastAgreed of the last block in the path and then sorts like PathWeight

12.8.3 Pruning Ghost Branches

- Since the Ghost Branches can never be a part of the Agreed.Block it allows the ghost branches to be pruned

  - This is due to the agreed parties being honest parties and since the ghost branch is not longer a ghost branch when it is seen by an honest party

12.9 Dynamic Parameters

12.9.1 General

- It was previously assumed that parameters like Tickets$_i$ and Votes$_i$ just were given

  - When the system evolves so must the parameters

  - e.g. in a proof of state system it is important to update the Tickets$_i$ since it is proportional to the money on the system

  - To avoid flickering when updating the parameters in can be done for instance each day

12.9.2 BlockParameter

- It turns out that it is convenient to require that dynamic parameters are a function of the parths in the tree and not a separate protocol

- **Definition 11.14 (block parameters)** The function BlockParameters is defined which at each block in the tree specifies what are the system parameters as compute at that block

  - Let Block be a block in Tree and Path = PathTo(Block) be the path from genesis to Block in Tree then the function BlockParameters(Tree, Block) is required to only be a function of Path

    * BlockParameters(Block) is written for brevity

- At each block it returns a structure Params = BlockParameters(Block) which contain the following entries:
    - Tickets is a dictionary which for a public key vk of a baker returns the number of tickets Tickets[vk] of the baker
    - FinalityCommittee
    - Votes
    - Hardness
    - Seed
    - SlotLength
    - ...
- For simplicity we can imagine that there runs a special designated smart contract on the blck chain, the **parameters contract**, which computes the new parameters
- The function BlockParameters would then just read state of this contracts
- The **block parameters** of a block are just BlockParameters(BLOCK)

- **Definition 11.15 (slot parameters)** A function SlotParameters is defined which at each slot specifies what are the system parameters as computed at last slot
    - It takes tree parameters SlotParameters(Tree, Block, slot)
    - It is required that Block has a slot number larger or equal to slot
    - Let Path = PathTo(Tree, Block) and $\text{Block}_{\leq \text{slot}}$ be the last block in Path with a block number that is $\leq$ slot then SlotParameters(Tree, Block, slot) = BlockParameters(Tree, $\text{Block}_{\leq \text{slot}}$
        - SlotParameters(Block, slot) for brevity
- **Definition 11.16 (advance parameters)** A function AdvanceParameters is defined which at each slot specifies what are the system parameters to be used at that slot

- It takes tree parameters AdvanceParameters(Tree, Block, slot)
- It is required that Block has a slot number larger or equal to slot − AdvanceTime
    - Then AdvanceParameters(Tree, Block, slot) = SlotParameters(Tree, Block, slot− AdvanceTime)
    - AdvanceParameters(Block, slot)) is written for brevity
- AdvanceTime should be large

- A new parameter is defined which is called StableTime
    - It is how often we change the parameters

Definition 11.17 (stable parameters) We define a function StableParameters which at each slot specifies what are the system parameters to be used for that slot. It takes three parameters AdvanceParameters(Tree, Block, slot). It is require that Block has a slot number larger or equal to slot − AdvanceTime. Then StableParameters(Tree, Block, slot) = AdvanceParameters(Tree, Block, slot − (slot mod StableTime)).

Definition 11.18 (final parameters) We let FinalBlockParameters(Block) = BlockParameters(PathTo(Block)[Block.LastAgreed]). We let FinalSlotParameters(Tree, Block, slot) BlockParameters(Tree, $\text{Block}_{\leq \text{slot}}$). We let FinalAdvanceParameters(Tree, Block, slot) = FinalSlotParameters(Tree, Block, slot−AdvanceTime). We let FinalStableParameters(Tree, Block, slot) : FinalAdvanceParameters(Tree, Block, slot − (slot mod StableTime)).

- Using final parameters of the tree layer has the advantage that you never get a rejected draw won by an honest party
    - The disadvantage is that if finalization turns off for a long time, the distribution of not follow the movement of money in the system which might not be healthy
    - The advantage of using nonfinal parameters is on the other hand that the tickets will follow the money no matter what happens to the finalization layer.
        - The disadvantage is that the advance time have to be set really large to reasonable account for worst case network
        - Might even a day

- A trick that gives the best of both worlds is to use two advance times WorstCaseAdvanceTime and FinalityAdvanceTime
    - FinalityAdvanceTime is small time which is basically the advance time we need to give the finalizers to get ready
    - WorstCaseAdvanceTime is very large
    - It uses FinalityAdvanceTime when computing FinalAdvanceParameters(Tree, Block, sl
    - It uses WorstCaseAdvanceTime when computing AdvanceParameters(Tree, Block, slot)

12.10 Some Important Missing Details

12.10.1 New Accounts

- When new accounts are created the account creator could pick the secret key sk_i such that $\mathrm{Draw}_{\mathrm{slot},i} = \mathrm{Sig}_{\mathrm{sk}_i}(\mathrm{LOTTERY}, \mathrm{Seed}, \mathrm{slot})$ is higher on average than the coming slot
    - Since the account creator knows the tree parameters it could just try several sk_i until it find a lucky one
    - To avoid this an account must use a seed published after it was created
    - A new account cannot take part in the lottery until a new seed was made
    - It is important that corrupted parties cannot influence the seed creation to much

12.10.2 Consensus Layer Peeking

- It is convenient to expose to the smart contract layer some value from the tree layer and the finality layer
    - This is done via peeking contracts
    - These are contracts in the smart contract layer which contains values from the consensus layer
    - A value V can be put ion the peek contract if it is guaranteed that all honest parties agree on V

        * They simply add it as a transaction to the next block and reject blocks that report a wrong value of V

    - An important peek is used to provide random values to the smart contact layer

        * These random values can be used to compute BlockParameters(Block)

        * Used to provide Seeds which provide fresh seeds for future lotteries

    - It can also be used to report values resulting from surveys run using Byzantine agreement

12.10.3 Training your Blockchain

- Finalization allows one to play with the parameters of your blockchain on the fly

    - You could try to lower the Hardness to tune the blocktime, to make the blockchain run faster.

        * If you go too low, the tree layer will start branching a lot, but the finalization layer will catch this since Δ will grow

    - You could increase the blocktime again until the branching goes away

        * Which will allow you to run with the lowest possible blocktime at all times.

        * In good network conditions it can be low, and when under attack it will become higher to protect the safety of the system.

    - This can all be controlled from the smart contract layer using the block parameter contract and

peek contracts

12.10.4 The CAP Theorem

- A network is **partitioned** if there are two parts of it that cannot talk to each other
- The **CAP theorem** roughly says that during a partition the system has to choose between availability and consistency

13 Network Security Mechanisms (12)

13.1 Authenticated Key Exchange

- Is about how to establish a secure channels based on an insecure network such as the Internet
  - The threat model throughout that the adversary can see and modify all communication
    - The entities being communicated with might be byzantine
  - It is assumed that two entities A, B wanting to estalish a secure wanting establish a secure connection both have certificates containing public keys pk_A, pk_B
  - The idea is now to use the public key pairs in some appropriate protocol to establish a short-lived session key K, which is then used to provide secrecy and authentication by secret-key techniques.
    - A motivation for this is efficiency since we do not want to encrypt large amounts of data using a slow public-key algorithm
- An **authenticated key exchange** protocol is a protocol for two parties A, B
  - Each party starts the protocol with the intention of establishing a key with some other party
  - At the end, each party outputs either "reject", or "accept" as well as a key.

- The protocol is said to be secure if the following three conditions hold:
    - **Agreement:**
        - Assume A intends to talk to B and B intends to talk to A.
        - Assume that both parties accept and A outputs key K_A, while B outputs key KB Then $KA = KB$
    - **Secrecy** and **Authentication**: if A wants to talk to B, and accepts
        - It must be the case that B participated in the protocol and if B also accepts
        - He did indeed intend to talk to A
        - The adversary does not know the key K that A outputs
        - A symmetric condition holds for B.
    - **Freshness** if $A(B)$ follows the protocol and accepts, it is guaranteed that the key K that is output is a fresh key
        - i.e., it has been randomly chosen for this instance of the protocol

13.2 How to not do it

- The following is an attempt at a authenticated key exchange protocol that doesn't work

1. A chooses nonce n_A and sends $E_{pk_B}(ID_A, n_A)$ to B.
2. B decrypts, checks ID_A, chooses nonce n_B and sends $E_{pk_A}(n_A, n_B)$ to A.
3. A decrypts, checks that the correct value of n_A appears in the result, and sends $E_{pk_B}(n_B)$ to B.
4. B decrypts and checks that the correct value of n_B appears in the result.
5. If the checks executed are OK, each party computes the session key as some fixed function of n_A, n_B.

- Example of why the protocol does not work

1. Suppose A starts a session with E, thus A will send $E_{pk_E}(ID_A, n_A)$ to E.
2. E decrypts this and starts up a session with B, pretending to be A. So E will send $E_{pk_B}(ID_A, n_A)$ to B.
3. B decrypts this, finds the right ID in the result, and sends $E_{pk_A}(n_A, n_B)$ to E.
4. E is of course not able to decrypt this, but can instead simply forward the message $E_{pk_A}(n_A, n_B)$ to A.
5. A will decrypt this, find a result that has exactly the form he expected, and will therefore return $E_{pk_E}(n_B)$ to E.
6. E can decrypt, find n_B, and is now able to send $E_{pk_B}(n_B)$ to B.
7. When B decrypts, he will accept since he finds the right value of n_B in the result.

- If there only runs one instance of the protocol there are no known attacks

13.3 The Secure Socket Layer Protocol

13.3.1 General

- One of most common solutions for authenticated key-exchange used today, is the **Secure Socket Layer protocol** (SSL)
  - It uses digital signatures in addition to encryption
  - The basic idea is that one party must sign a nonce chosen by the other
  - SSL is used to set up secure http-connections on the Internet
    - It is placed between the application and the TCP/IP transport layers
    - The data that it transports are put inside TCP/IP network packets
    - The contents of the packets can only be handled by SSL compliant clients or servers
  - It is always executed between a Server and Client
  - In its basic form, it requires that both Server and Client have public-key certificates and has the corresponding private keys.
    - They must be able toverify each others' certificates

  - It is also possible to do a one-sided SSL where only the server has a certificate
    * Common use case in practise

- SSL is composed of serval protocols
  - **Record Protocol:** is responsible for the "raw" transmission of data
    * Other protocols rely on this one for transporting data
    * It uses a so called *cipher spec* to determine how to encrypt/decrypt and authenticate / verify data
      · It say which encryption and other algorithms to use for thi
      · It may be empty which means no encryption should be used
      · The typical case is that encryption and authentication algorithms are specified and client and server store keys for both purposes
    * It processes a packet by
      1. Adding a sequence number and a message authentication code to the raw data
      2. This entire string is encrypted.
      3. The receiver expects to receive packets in numeric order and halts if this does not happen.
    * This prevents replay and works because we rely on TCP/IP to make sure that packets will often arrive in the order they were sent
  - **Handshake Protocol:** is the pars of SSL that does authenticated key exchange
    * The job of it is to bring the server and client from a state where they have an epty cipher spec and not common keys to a point where they have negotiated which algorithms to use and have done an authenticated key exchange

        * The client initially sends to the server a list of the cryptographic algorithms it supports in decreasing order of preference
        * The server then says which of the possibilities it has selected
        * Then the actual authenticated key exchange is done
        * The parties exchange Change Cipher Spec messages signal that we should now start using the negotiated algorithms and keys
    - **Change Cipher Spec Protocol**: is a one message protocol where one party tells the other to change from one cipher spec to another that has just been negotiated
    - **Alert Protocol:** is a one message protocol used to signal error messages to the other side

- It is important to understand that even if we verify theoretically the security of the actual key exchange protocol, it does not guarantee security of the entire SSL construction.
    - An adversary may try to manipulate messages between honest players to try to make them use an algorithm for encryption that is weaker then what they can actually use
        * Possible since while we are doing the handshake, the communication is not yet protected.
    - Another issue is that we should of course do a Change Cipherspec just after the Handshake, so we can start using the session keys we just established.
        * If this is not done the messages might be send in the clear

13.3.2 The SSL key exchange

- This is a description of the main part of SSL, the actual key exchange protocol with many detail stripped away

1. C sends a hello message containing nonce n_C.
2. S sends a nonce n_S and its certificate $Cert_S$ (containing the public key pk_S of S).
3. C verifies $Cert_S$, and chooses a so called pre master secret pms at random. C sends $E_{pk_S}(pms)$ to S, also C sends its certificate $Cert_C$ to S, plus its signature sig_C on the concatenation of n_C, n_S and $E_{pk_S}(pms)$.
4. S verifies $Cert_C$ and sig_C. If OK, it decrypts pms.
5. S sends to C a "finished" message containing essentially a MAC on all messages he has sent and received sent in this instance of the protocol, with pms as the secret key.
6. C verifies the MAC, and if OK returns its own finished message to S, also with a MAC on all messages sent and received up to this step (note that this now includes the finished message from S, so one cannot just repeat the previous message). S verifies the MAC when it is received.
7. At this point both parties derive from n_S, n_C and pms a set of keys for secret-key authentication and encryption of the following data exchange. Typically this is done using a hash function.

- The idea is that S authenticates itsself by being able to send a finished message with a correct MAC
  - This proves that it was able to compute pms and hence knows sk_s
  - The client authenticates itself by being able to sign a message containing the encryption of pms
  - C establishes that it knows the plaintext by sending a correct "finished" message at the end
  - MACing the entire view ensures that each party can check that the other party had the same view of the protocol
    - i.e. the attacker did not change any messages
    - It forces the attacker to only look at messages sent by the peers and forward these truthfully

13.3.3 Forward Secrecy and the Future of SSL/TLS

- **Forward secure** means that the security of the session cannot be comprimised even if the adversary steals a long-term secret key some time in the future
  - A variant of the SSL handshake that is based on RSA is not forward secure

- A variant of the SSL handshake that is based on Diffie-Hellman is forward secure

13.4 IPSec

13.4.1 General

- **IPSec** is a set of protocols that do a job very similar to what SSL can do, but on the transport layer
  - It is a set of protocols for setting up a secure connection
    - Also known as a **security association** between two ip adresses
  - Since it is on a lower layer it means that the sequence numbers and other control information are encrypted while in transit which is not the case of SSL
  - To create a security association, it uses a key exchange protocol known as the Internet Key Exchange (IKE)
    - It is a name for a rather large set of different alternative methods.
    - Public-key certificates are used to identify the other party, just like in SSL
    - The public-key technique that is used, is known as the Diffie-Hellman key exchange

13.4.2 (Authenticated) Diffie-Hellman Key Exchange

- **Diffie-Hellman key exchange** in its basic form uses
  - A arithmetic modulo a prime number p.
  - A number g in the interval from 0 to $p1$ is chosen once and for all
  - When client C and Server S want to exchange a key, we do the following:

1. C chooses a random number a and sends $g^a \bmod p$ as well as his certificate to B.
2. Likewise, S chooses b at random and sends $g^b \bmod p$ as well as his certificate to A.
3. C computes $(g^b \bmod p)^a \bmod p$ based on what he got from S and his own random choice. Likewise, S computes $(g^a \bmod p)^b \bmod p$. These two values turn out to be equal, namely they are both equal to $g^{ab} \bmod p$, so this value can be used as a common key.
4. C signs all messages he has seen so far and sends the signature to S. Likewise S signs all messages he has seen and sends the signature to C.
5. Both parties verify the received signature against the messages they have seen and the public key of the other party. If everything is OK, the protocol was successful.

- If p is chosen large enough, it is believed to be infeasible to compute from this information the key gab mod p.
  - It is the so called **Diffie-Hellman problem**.
- Issues with Diffie-Hellman key exchange
  - Issue arise when the protocol is based on arithmetic modulo a prime
  - The problem comes from the fact that there is an algorithm known as Index Calculus that will solve the Diffie-Hellman problem, and hence break the protocol.
  - The special feature is that it will be able to do so very efficiently, if one first does a very long and inefficient preprocessing, that only depends on the prime p.
    - It means that once the preprocessing is done for a particular prime p one can easily break any instance of the protocol that uses that prime p
  - There are several ways to solve this problem:
    - Either a fresh new prime for the Diffie-Hellman protocol should be chosen, if not for every instance of the protocol, then at least with regular short intervals
    - A different setup can also be used, namely the Diffie-Hellman protocol can be based on so-called Elliptic Curve cryptography instead of arithmetic modulo a prime

13.5 Comparison of SSL and IPsec

- For IPsec, the communicating parties will be units with an IP number, for a standard PC or laptop, this would be the network adapter
  - This means that the "secure tunnel" goes between such IP nodes and as soon as the network adapter delivers data to the machine it sits in, the data is no longer protected
  - All applications on the machine can use the protected connection without being aware of the encryption
- For SSL, the communicating parties are applications, typically a browser and an Internet server
  - This means that the applications need to know how to run SSL
  - Data are protected all the way from inside the client until it reaches the server
  - On a multiuser machine, for instance, this can be used to separate applications owned by different users.
- Both SSL and IPSec establish secure tunnels between two entities, after the data leaves the tunnel, it is no longer protected.
  - If what your application requires is signatures on documents, then neither SSL not IPSec are useful.
  - The situation is the same is you want to store data in encrypted form
    - Since data are not protected when it leaves the tunel

13.6 Password-authenticated key exchange

- As mentioned, one flexibility offered by SSL is that one can allow for “one-way” authentication
  - The server identifies itself towards the client but no authentication is performed of the client
  - It is the best you can do when the client does not have a certificate

- When a secure connection is established the client can now authenticate itself by sending a password to the host over a secure connection
    - Since the server is authenticated and the channel is encrypted, the client knows it will not be revealing the password to a malicious server.
- The "one-way SSL plus password" method is useful, but also has some disadvantages
    - It only allows the client to authenticate itself towards the servers it holds passwords for
    - It becomes harder to prove security of the total process because the sending of the password is not an integrated part of the protocol
- Since guessing the password attack is an attack can never be prevented, the optimal result is to prove that this is the best possible attack.

- **Password authenticated key exchange protocols** are relatively new, but one such protocol, known as **SRP**, has been standardized
    - It has no security proof curerently
    - We base ourselves again on Diffie-Hellman, but with a twist, namely we encrypt the messages under the password pw that is assumed to be known by both client and server.
    - The description of the protocol below assumes a successful run where both the client and the server have the same password:

1. C chooses a random number a and sends $A = E_{pw}(g^a \bmod p)$.
2. Likewise, S chooses b at random and sends $B = E_{pw}(g^b \bmod p)$.
3. C decrypts B with his password pw and computes $(g^b \bmod p)^a \bmod p$ based on the result of the decryption and his own random choice. Likewise, S computes $(g^a \bmod p)^b \bmod p$. As mentioned before, these values are equal and can be used as a common key.
4. Here follows some steps where the parties confirm that they agree on the key and have seen the same set of messages. The details are not important here.

- If the passwords used by the client and the server in Step 1 are different, then the protocol will not succeed
  - Decrypting a ciphertext with the wrong key can be typically assumed to produce just random garbage
  - The intuition for the security behind this construction is that an adversary who listens to an execution of the protocol should not be able to brute-force the password just from the transcript of the protocol
  - This protocol requires both parties to know the password in cleartext. But in most cases the server does not store the password, but instead a hash value or similar computed from the password
    * SRP and related protocols are able to handle this case as well; they can work even if the server has some encrypted or hashed representation of the password.

13.7 Applications of Secure Channel Set-up

13.7.1 Secure http

- Secure http is simply another name for usage of SSL or TLS to set up a secure connection between a web server and a client
  - It can be referred to directly in web addresses as `https://www....`
  - Such a connection works on a level below the application
    * It creates a secure tunnel between the client and the server
    * Anything that goes throught is encrypted and checked for authenticity and the higher level application does not need to be aware that the connection is secure.

13.7.2 Virtual Private Networks

- Virtual Private networks (VPN) solutions are designed to set up secure connections from a local area network to a remote PC.
  - During the set-up, which happens between the PC and a VPN gateway, the remote PC is assigned an IP-address as if it was a part of the local network.

  - After setting up the connection all traffic betweenthe PC and the VPN gateway is encrypted and checked for authenticity
  - It requires, of course, that an authenticated key-exchange is done during the set-up
  - The PC can now work as if it was physically a part of the local network.

13.7.3 Higher Level Channels

- There are several protocols and solutions around that provide security for higher-level objects. For instance, the S/MIME standard
  - MIME is the standard format in which emails are transported on the net
  - S/MIME is an extension where some extra fields are added, that allow transport of encrypted and signed emails
  - Another way to do "high-level" cryptography is the Pretty Good Privacy (PGP) software
    * It provides capability to sign and encrypt mail and documents
  - It does not use certification of public keys by CA's, instead it uses a concept known as "web of trust"
    * i.e., your friends can send you a public key of a third person and assert that they trust this public key

14 System Security Mechanisms (13)

14.1 The Trusted Computing Base

- **System Security** is about defensive mechanism to try to prevent external attackers from getting into the system
  - Also about preventing attacks from parties who are already users of the system

  - The typical scenario is that some party wants access to some part of our system, and we want to ensure that access is only given in cases where this is OK w.r.t our security policy
  - Getting access to a machine does not necessarily mean physical access, but could also just mean that I am allowed to communicate with it
    * Identification is not enough it also needs to have some part that takes the decision on whether some action should be allowed according to the security policy
    * If access is denied it should not possible to circumvent the system and get access anyway
  - There needs to be a part of the system that cannot be modified by attackers
    * It is called the *trusted computing base* (TCP)
    * It can e.g. be established using hardware and/or software security depending on the threat model
    * It will often be a part of the operating system's kernel
    * It consists of one or more hardware units

14.2 Firewalls

14.2.1 Packet Filtering

- A firewall is simply a machine or a piece of software that stops and removes part of the traffic on a network
  - It is typically placed on the connection between a company's local network and the Internet
  - The simplest use of a firewall is to make it filter out all communication trying to connect to machines for which you do not want any communication with the outside world.
- The simplest tool we can use in a firewall is called *packet filtering*

- The firewall looks at each packet and decides whether the packet should be allowed through based only on the content of the packet and some fixed set of rules
- It can e.g. remove all packets trying to open a TCP connection to a local machine and stop all UDP traffic
- If one would have to make the firewall allow for incoming connections to e.g. the web-server or to allow users to use UDP
    - It can be done by packet filtering, because we can determine which incoming connections is for the web-server, and which UDP packets are meant for or are coming from a known DNS server.
    - It opens up for attacks if the attacker can break into the web-server and move from there to other machines
    - It can be solved by e.g. using two firewalls
- If simple packet filtering is to be secure, it needs to be so restrictive that it is hard to live with
    - If we make it less restrictive, we open a possibility that an adversary can try to communicate with and attack machines on the local net
    - An attack is typically done by scanning through a large interval of IP addresses until one findsa machine that is willing to respond.
    - It is known as "port scanning"
- The conclusion is that a firewall needs to do something more intelligent than just a simple packet filter
    - It needs to examine packets in more detail and either take action that depends on the purpose of a packet, or even modify the packets.

14.2.2 Proxy Firewalls

- One radical approach to "intelligent filtering" is a **proxy firewall**
  - It will not allow any connection at all from a local machine to the outside or vise versa
  - It will handle any connection needed on behalf of the local machines
  - If a local machine has some client software that wants to use a service on the net the client must be configured to connect to the firewall instead and then the firewall connect to the service
  - The local network is completely hidden form the outside
    - Scanning through a large number of IP addresses will not work
    - You'll never get a response except from the firewall
    - You are forced to hack the firewall before attempting anything else
  - They are regarded highly secure
  - It is not a very flexible solution
    - If we want to use client software for some web service on a local machine this is not possible
    - We will be forced to do some special implementation or configuration for every type of connection

14.2.3 Stateful Firewalls

- A **stateful firewall** keeps track of every connection going through it
  - Whenever someone tries to open a connection it checks whether this type of connection is allowed and blocks if not
    - There need to be some flexibility for UDP packets

        * This requires the firewall to remember which connections are active at the moment

    - It is even possible to translate the addresses used by local machines to something different in the outside traffic

        * This hides the local addresses from an outside attacker
        * Know as a masquerading firewall
        * It makes the IP address scanning harder

    - A final step is to have the firewall inspect the behavior of a connection to see if it does something suspicious and either stop it or drop packets that have suspicious content

        * e.g. in some attacks on web server the attacker would need to send an unusually long input string to a particular routine running on the server
        * The firewall can be configured to catch such traffic and remove it

    - The optimal configuration of the firewall of course depends on the configuration of the local network

        * It to be updated as the structure of the local network changes

14.3 Malicious Software and Virus Scanners

- One of the things attackers would want to do it they make it past the firewall is to install malicious software on the attacked machines. It comes in many forms

    - **Trojan horses:** programs that appear to be useful, or goes completely unnoticed by the user, but have some hidden malicious purpose

        * Such as granting unauthorized access, reveal private data or destroy data

        * A well-known special case is Spyware: programs that monitor peoples' Internet usage and send information on this to the creator of the program

    - **Viruses:** infect programs on your computer, spread themselves from one program to another, and eventually does something harmful to the machine

    - **Worms:** similar to viruses, but exist as stand-alone programs that actively try to spread themselves from one machine to the other

- **Virus scanners** are used to find malicious software that makes it into your computer

    - It can be used to scan all potentially dangerous files and locations on your harddisk looking for pieces of code that is known to be suspicious

    - It can use a database containing information on a large number of known viruses

        * This is typically in the form of **virus signatures** which is particular bit patterns characteristic for particular viruses

    - They are only useful if their databases are kept up to data

        * The largest producers of virus scanners are typically remarkably fast in updating their data files when a new virus hits the net

        * Many users do not update their own copy of the scanner, or do so too slowly

        * Infections happen because of outdated virus scanner files and not because of new viruses

    - Virus's try to fool the scanners databases by encrypting most of their code and keep a small piece of cleartext code that will decrypt the real virus

- There are several ways an attacker can get malicious code on your machine
  - The attacker can try to exploit weaknesses in the operating system or server software
    - e.g. overflow attacks or similar
    - This can lead to attacks that will work if a user just visits a particular web page, provided his operating system has some known and exploitable bug
  - In many cases it is easy to fool the user into installing the malicious code himself
    - This is a special case of social engineering

14.4 Intrusion Detection

- A different way to spot malicious software of users is to try to detect malicious behavior instead of malicious code
  - Process and/or users are being monitored to try to detect if their behavior is suspicious
  - The advantage is that it does not depend on information on already known viruses and/or attacks
  - e.g. stateful firewalls
- There are two approaches to deciding whether something suspicious is going on
  - **Rule-based intrusion detection:** We set up a set of rule describing what normal behavior is
    - Anything that is significantly different is considered illegal
  - **Statistical intrusion detection:** First gather statistics on actual honest behavior and then compare to the statistics of the observed user or program

- The basic problem with intrusion detection is that the entity monitoring the system must be tolerant enough to allow legal behavior, but restrictive enough to catch suspicious behavior
  - Another issue is that the entire idea is no good, if the system does not react when an attack is detected
- A final very useful trick in the intrusion detection area is the so called *"Honey-pot"* approach
  - The idea is to create a resource on your system that will look interesting to an attacker but is actually fake
    - A resource that no honest user would want to use
  - As a result, if activity involving the honey-pot is detected, we can assume this is adversarial activity
  - By logging such traffic one can hope, for instance, to trace it back to the attacker before he becomes aware that he is being traced

14.5 Security Policies

14.5.1 General

- **Definition 13.1** A **security policy** contains a description of the system that the policy applies to
  - It describes the security objectives you want to achieve
  - It can take different forms
    - It can be a specification of what different entities are allowed to do and not allowed to do
    - It can be a specification of which events we want to tolerate when the system runs
  - The system must then ensure, either that no entity can do what is not allowed, or that no event occurs that we do not want to tolerate

        * The policy may contain a high level (non technical) description of the methods that will be used to enforce the security objectives

    - It is important to say which system we want to protect

        * This has the purpose of limiting the problem

        * We do not have to protect something that we have explicitly defined to be outside the system

    - The security policy can be thought of as the place where we formulate the rules that the TCB should enforce.

14.5.2 Models for Security Policies

- **Models for Security Policies** An abstract description of ways to design a security policy

    - It does typically not describe a particular system

    - It often addresses particular threat models

- **Security Policy** Is typically a realization of some model with respect to a particular concrete system

    - It can also be based on more than one model

    - It can also not be based on any model and therefore "unique"

- A lattice consists of a finite set S plus a relation $\leq$. For any elements $a, b, c \in S$ it must hold that

    - $a \leq a$

    - $a \leq b$ and $b \leq a$ implies $a = b$

    - $a \leq b$ and $b \leq c$ implies $a \leq c$

- One may think of S as a set of different privileges or rights a user may have in a system

- $a \leq b$ means that with rights b, you can do at least as much as with rights a
- It may be that some rights are not comparable
    * Therefore the definition of a lattice does not require that the relation $\leq$ is defined for all pair a, b

- In order for $S, \leq$ to be a lattice, it is required that for any $a, b \in S$ we have
    - There exists a **greatest lower bound** for a, b
        * i.e. there is some $c \in S$ such that $c \leq a$, $c \leq b$ and for any other c' with this property, we have $c' \leq c$
    - There exists a **least upper bound** for a, b
        * i.e. some $d \in S$ such that $a \leq d, b \leq d$ and for any other d' with this property we have $d \leq d'$

- There are (at least) to ways to use the lattice model to organize a security policy
    1. Define **subjects** to be all users and processes in a system, which is any entity that may actively do something to other parts of the system
        - **Objects** are resources, files, hardware devices etc.
        - We classify subjects and objects such that each of them are assigned some position in a lattice
        - A subject s with classification $C(s)$ may do a certain operation on object o with classification $C(o)$ if a and only $C(s) \geq C(0)$ or is and only if $C(s) \leq C(o)$
    2. We can assign a position in a lattice to different parts of a system
        - We may say that information may only flow from position a to position b if $a \leq b$

        * This would mean that higher positions are "more secret"
        * The policy focuses on confidentiality
    - If a policy focuses on authenticity higher positions would mean "more reliable parts of the system"
        * We would specify that information can only flow from a to b if $a \geq b$

- The first item above gives a motivation for requiring in the definition of a lattice that least upper bounds and greatest lower bounds exist.
    - e.g. if we have two files, classified as a, respectively b, then all users with classification least upper bound of a, b or higher can read these files

- A model simply special case of the Lattice model is the **Bell-Lapadula model**
    - The idea is to define a number of security levels, which are linearly ordered
        * e.g. public $\leq$ secret $\leq$ top-secret
    - The model defines to rules
        * **No read up:** Subject s may read from object $C(s) \geq C(o)$
        * **No write down:** Subject s may write to an object o only if $C(s) \leq C(o)$
    - The idea is to prevent information from flowing from higher levels to lower ones
    - It is very rigid and hard to handle in practice
        * The system may not be able to do what it is supposed to unless some information flows downwards

- The **Biba model** is similar to Bell-Lapadula but focuses on integrity

  - The higher levels are the ones that contain the most reliable information
  - It has two rules
    * **No read down:** Subject s may read from object o only if $C(s) \leq C(o)$
    * **No write up:** Subject s may write to object o only if $C(s) \geq C(o)$
  - The idea here is that information must not flow from lower levels to higher ones
    * It ensures that highly reliable information is not "contaminated" with less reliable stuff

- Examples of models that re not directly based on lattice structures
  - **Chinese Wall** This is a model inspired by commercial scenarios
    * e.g. a company C has several different clients
    * Some of these clients are competing with each other
    * A predefined predicate *compete* is assumed
      · $compete(c_1, c_2)$ for clients c_1 and c_2 is true if and only if c_1, c_2 are competitors
    * The rule is that subject s is granted access to information about c if and only if he never accessed information on any c' for which $compete(c, c')$ is true
      · This rule is dynamic since as soon as s has accessed information on c this prevents him from accessing information from any competitor of c
  - **Prevent-Detect-Recover** The idea is to split the policy in 3 parts
    * One part describing how we will try to prevent attacks from happening at all

        * A part describing how we can detect a successful attack
        * A part describing how we will recover once we detected a breach of security
    - **Separation of Duty** is a model that comes in two flavors:
        * **Dual Control** is a model where certain action are defined to be possible only if they have been authorized by several persons or entities
            · A well known example is the use of nuclear weapons
        * **Functional Separation** is a model where an action requires involvement of several persons or systems, but at different points in time
            · e.g. signing up for an exam

14.6 Access Control

14.6.1 General

- It is implicitly assumed that when a users wants to execute an operation on some object able to verify the identity of the user and figure out which rights and privileges he has
    - Checking the identity of the users and figuring out which rights and privileges he has is a technical problem
    - Figuring out which rights the user has is a different problem
        * In principle, this can be computed from the 's identity and the security policy
        * Doing so on the fly while a user is waiting for access, for instance requires that the information about this is organized in a appropriate way
- The generic way to represent the information is known as the **Access control matrix**
    - It is a matrix A with a row for each subject s and a column for each object 0

  - Entry $A[s, o]$ contains a list of all operations s is allowed to execute on o
  - On realistic systems storing the entire matrix in a central location is not practical

- In practice two basically different approaches are taken
  - **Access Control Lists** Here we store, together with each object the column assigned to the object
    * This is called an access control list (ACL)
    * Such list make it easy to decide for a given object who has access
    * It is much harder to say what a given subject can do
    * This is basically the approach of the UNIX system
  - **User Capabilities** Here we store subject, the row assigned to the subject
    * This is called a user capability
    * They make it easy to compute what a given user can do
    * It makes it harder to decide who has access to a given object
    * Windows uses this basic approach

- For large systems, ACL's gets too long if they are specified per subject
  - One typically splits subjects into predefined groups
  - The ACL says only what subjects from each group can do
  - UNIX splits subjects from the point of view of a single user, into: the user himself, the user group he is in, and anyone else
    * Groups also make it possible for several subjects to share the same list of user capabilities

- A variant of the user group idea is known as **role-based access control**
    - A number of roles are defined, such as "ordinary user", administrator", etc. and the capabilities are specified for each role
    - Users are assigned a certain role when they log in and are granted access according to the role

14.6.2 When the Access Control Matrix is Dynamic

- There are basically two approaches for updating the access control matrix *mandatory access control*, where subjects cannot change the matrix, and *discretionary access control* where subjects have the power to update the matrix, or parts of it
    - Given that access control matrix may change over time it is natural to ask what types of information flow this may allow
    - e.g. can a certain access right r ever be added to some cell $A[s, o]$ that did not contain r before
    - Harrison, Ruzzo and Ullman have studied this question in a formal model, where certain primitive operations on access control matrix A are allowed, namely
        - Create/delete subject s
        - Create/delete object o
        - Add/delete access right r from $A[s, o]$
    - It is assumed that there is a fixed finite number of different access rights
    - A command has the following format:

```
Command com(parameter-list)
r1 is in A[s1,o1]
..
rn is in A[sn,on]
list of primitive operations
```

- The meaning is that the list of primitive operations will be executed if all the conditions in the first part are satisfied
  - All the rights, subjects and objects referred to in the specification of `com` are taken from the parameter list
  - If one is given a fixed set of commands $com_1, \ldots, com_t$ and an initial access control matrix one can ask for given access right r, does there exist a set of commands that will transform A into a matrix A', where $r \in A'[s, o]$ for some s, o but where $r \notin A[s, o]$
    - If the answer is no A is safe with respect to r
  - One may want to prove
    - If commands may contain more than one operation, then it is undecidable whether A is safe w.r.t. r.
    - If all commands only contain a single primitive operation, then it is decidable whether A is safe w.r.t. r.
    - If there is a upper bound fixed ahead of time on the number of subjects and objects, then it is decidable if A is safe w.r.t. r.
- Therefore the conclusion is that when designing dynamic access control sys tems, the best advice is: keep it simple!

15 Attacks and Pitfalls (14)

15.1 Categorizing Attacks

15.1.1 General

- A way to characterize what a attack based on what the attacker is trying to achieve (**STRIDE**)
  - **Spoofing Identity**: results the attacker being able to impersonate another person (user)
  - **Tampering**: results in attacker being able to manipulate data without being detected

  - **Repudiation**: results in the attacker being able to deny doing something he actually did
  - **Information Disclosure**: results in the attacker being able to access data he should not see
  - **Denial of service**: results in the attacker being able to deny others access to a system
  - **Elevation of privilege**: results in the attacker being able to have more rights that he should have

- The same attack might have several different effects at the same time
  - Depends on the context

15.1.2 What means are used in the attack: X.800

- A way to characterize attackers are according to the means utilized by the attacker
- The X.800 standard operates with two main types of attacks on network transmission
  - **Passive Attacks:**
    * **Eavesdropping:** The attacker listens in and looks at the information sent
    * **Traffic Analysis:** The attacker only looks at who is sending to who and how much is sent
  - **Active Attacks:**
    * **Replay:** The attacker resends an old message
    * **Blocking:** The attacker stops a message from arriving
    * **Modification:** The attacker changes the data sent or injects a new message of his own
- Passive attacks are very difficult to detect

  - A defense it to use encryption against eavesdropping
  - Encryption cannot stop traffic analysis
  - Ways to avoid traffic analysis
    * Send the same message to many parties to hide intended receiver
    * Randomize the length of the message to hide the size of the actual message
  - A message known as stenography can be used to hide information inside a message
    * e.g. make slight changes to a digital picture to hide a message
- Active attacks are easier to detect
  - The focus on detection of attacks, using MACs, signatures, sequence number of message etc.

15.1.3 Where are we attacked, an by whom: EINOO

- **Who** attacks us
  - **External attackers:** who are not legal users of the system
  - **Insiders:** Who are registered as users of our system and so have some amount of access to start with
- **Where** the attacker is able to hit us
  - **Network attacks:**
    * The adversary can only listen and perhaps modify network traffic
    * Limited to the means described in X.800
    * It is usually impossible to prevent that network attacks are attempted but most of them can be defended against against using cryptography

- **Off-line attacks:**
    - The adversary gets unauthorized access to information that is stored permanently in the system
    - He can steal and/or modify the information
    - Such as stealing the password database file
    - Harder than network since they require to break the systems access control
    - Cryptographic protection of data is a obvious defense against off-line attacks
        - Since the keys has to be stored somewhere an if the adversary has access to this, this protection is then useless
- **On-line attack:**
    - The adversary breaks into the system while programs handing sensitive information are running
    - He may be able to read secret keys from RAM
    - They are harder to mount than off-line
    - The adversary needs to completely break the systems access control
    - They can have very severe consequences and can only be defended against by having different parts of the system protected in different ways

- The EINOO classification can be useful when designing threat models for concrete systems

15.1.4 Where did we make the mistake: TPM

- A final way to categorize attacks is according to what makes the attack possible

  - **Threat model:** The attack is possible because the thread model was not complete
    * An attack not thought about turned out to be relevant
  - **Policy:** The attack is possible because the specification of the security policy expresses something different from what we intended
  - **Mechanism:** The attack is possible because the security mechanisms can be circumvented

15.2 Examples of Attacks

15.2.1 Illegal Inputs

- The most important type of attack in practice is the type that circumvents security mechanisms by exploiting mistakes in the software that supposedly implements the mechanisms.
  - It takes a number of different forms
  - The attacker gives some input to the program where the input has a form that the programmer did not expects
  - It may cause the program to behave in some inappropriate way that the attacker may be able to explid
  - Usually happens when the data is interpreted as something else than just data
- In an **overflow attack** the adversary gives input to a program that is longer than expected
  - If the programmer did not check for this properly some bad things can happen
  - Programs written in C may have this problem
  - C does not have any built-in protection against many kinds of run-time errors such as writing outside an array
- **The Unicode Exploit**

  - The IIS is supposed to check whether the requests it gets from outside try to access dictionaries that should not be accessed
  - Requests are allowed to contain unicode characters
    * This allows directory names to contain all kinds of international characters
    * They must be decoded before taking action
  - The IIS previously did the security check before the decoding
    * This allows the attacker to mask an offending request by encoding it in unicode in a special way that the security check will not recognize
    * The attack consists of sending carefully masked request to the ISS
    * This would run with same rights as the IIS
    * The attacker can upload an run various interesting software to the target computer

- In **Cross-Site Scripting**, the attacker exploits poor design of web pages
  - A web site might be vulnerable to cross-site scripting if it has a page that accepts input from a client and then echoes this input back to the user in a new page
  - It is possible to fill in the search block with executable JavaScript code
    * When the string comes back the browser will execute this code
    * This can be used to fool other users of the system

- **The Heart Bleed Bug**
  - Occurred in OpenSSL which is an open-source software package that implements SSL/TLS

- The client can confirm that the server still is alive
    - This is done by sending a nonce to the server and have the server return the same nonce
    - The client sends both the nonce and its length
- The server has at some point store in RAM both the nonce and the length ℓ
    - It will produce the message to the client by reading ℓ characters from the array
- The Heart Bleed bug was a simple case of missing input validation
- If a malicious client sends a length that is much too large, the server will continue to read beyond the end of the array and will return some piece of its internal memory to the client