Distributed Systems

Satvik Gupta

January 28, 2023

These notes are by no means comprehensive or complete

Nodes on the internet are identified by IP addresses. Data is passed from one router to another, moving each packet closer to its destination, in the hope that it will ultimately be delivered.

Each router must regularly be supplied with up-to-date routing tables. Individual IP addresses are grouped into prefixes. Autonomous Systems (AS) own these prefixes. Routing tables between ASes are maintained using Border Gateway Protocol (BGP).

Autonomous Systems

An Autonomous System can best be defined as a collection of IP routing prefixes, that are under the control of 1 or more network operators, on behalf of a single entity or domain, that **present a single,common and clearly defined routing policy to the Internet**.

- Each AS has a number (ASN).
  - There may be ASes without ASNs too. It is possible that an organization is running BGP using private AS numbers given by an ISP.
  - The ISP must have an officially registered ASN. The multiple private AS numbers are supported by the ISP.
  - The Internet, however, can only see the routing policy of the ISP. It's the ISP's duty to include the routing policies of the private AS numbers in its routing policy.

Transit vs Peering

Peering

Peering is when two ASes allow free-flow of traffic between them and their downstream customers. Peering relations are free of charge. Peers cannot see each other's upstreams.

Transit

Transit is when an AS (provider) agrees to:

- Direct the traffic of another AS(customer) to the internet
- Direct traffic from the Internet to the customer AS.

A transit fee is charged by the provider to the customer.

- The transit provider gives the customer a list of routes that belong to other ASes/ISPs that the provider is connected to. The transit client can send/receive traffic from those routes through the provider.
- The transit provider also advertises its customers' routes to its connections, so they know that traffic to those routes needs to go via the transit provider.

The transit provider may themselves use other transit networks too.

A transit-free network is a network based only on peering. These are generally the top-level ASes. They provide transit to smaller ASes, like ISPs.

IMPORTANT

A TRANSIT PROVIDER WILL NOT ANNOUNCE A PEER ROUTE TO OTHER PEERS, OR TO NETWORKS IT BUYS TRANSIT FROM

- If it did, it would be providing free transit over its network to its peers.
- Or worse, buying transit and giving it away for free to peers.

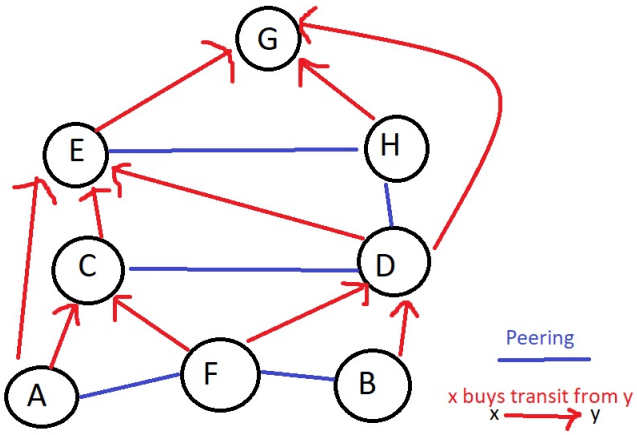

Figure 1: Transit and Peering Example Diagram

In the above represented network,

- G can see all networks since E,H and D buy transit from it.
- A can see F and customers of F, but it cannot see B through F.
- C can see B through D, but not through F.
  - D will want to tell B that it has access to C, to provide more incentive to B to buy its transit. After that, it will be obligated to let C know that it has access to B, since B bought transit from D. Therefore, C will see B through D.
  - F will not let C know it has access to B. If it did, F would have to pay for C->F->B transit.
- A can see B through C, but not through F.
- Traffic from C->H flows through E, but not through D.

BGP Hijacking

BGP was designed without security - it is based on trust. If an AS says it controls certain IPs, all its peers will believe it, and route app traffic to those IPs, to that AS. Security extensions and third-party validations exist, but they're not widely used.

An AS may announce an IP it doesn't own, or claim to have a shorter path to it than is already available - even if that "shortest" path doesn't actually exist.

Generally, ISPs filter BGP traffic. They allow BGP advertisements from their downstream networks to contain only valid IP space (i.e, IP addresses that the downstream network is known to possess). However, hackings have occurred because this isn't always true.

If an AS is hacked, this can be exploited. However, this will be quickly found out and reversed.

OSI Model

Open Systems Interconnection.

Widely referenced in academic literature, not often used exactly in industry. For example, the TCP/IP protocol doesn't fit neatly on top of the OSI model.

From lowest to highest, the 7 layers are:

1. **Physical Layer**

   Deals in bits. Handles transmission and reception of raw bit streams over a physical medium.

2. **Data Link**

   Deals in frames. Transmission of data frames b/w 2 nodes. Handles frame sync, errors, QoS, physical addressing, etc.

   - Physical Addressing, for e.g, using MAC addresses.
   - NOT Network Addressing.

3. **Network Layer**

   Deals in packets. Handles structuring and managing a multi-node network, including addressing, routing,etc.

4. **Transport Layer**

   Deals in segments/datagrams.

   Handles reliable transmission of data segments b/w points on a network. Handles segmentation, acknowledgment, multiplexing, handshakes, etc.

   - Segmentation is dividing large data into smaller sizes in order to match packet size imposed by network layer. This is known as MTU (Max Transmission Unit).

5. **Session Layer**

   Manages communication sessions, i.e, continuous exchange of information in the form of multiple back and forth transmissions btw 2 nodes. Creates the setup and controls the connection. Performs teardown. DNS is often put in this layer.

   Logon, name lookup, log off occur here.

   Authentication in FTP is built into session layer.

6. **Presentation Layer**

   Translates data b/w a network and an application. Includes character encoding, data compression and encryption/decryption.

   AKA Syntax layer.

   TLS/SSL is generally considered to be in this layer.

7. **Application Layer** High level protocols, such as HTTP or FTP. Generally include file sharing, message handling, DB access, etc.

The Internet Protocol Suite

The Internet Protocol Suite (TCP/IP) is different than the OSI model. Many layers are similar, but their differences start after the network layer.

TCP/IP layers don't fit neatly into OSI layers.

Layers in TCP/IP suite are:

1. Physical
2. Data Link
3. Network

4. Transport
5. Application

TCP/IP doesn't differentiate between session,presentation and application layer.

This is why it doesn't fit correctly into OSI.

For example, TLS is built on top of transport layer such as TCP. But, applications generally use it as if it was a transport layer.

Some mentions of TCP/IP don't even contain a physical layer. The Data Link and Physical layers are merged into a common *"Link"* layer. It is unspecified as if the Link layer does the job of the physical layer, or if TCP/IP assumes that physical hardware already exists underneath.

Application Based Multicasting

Nodes organize themselves into an overlay network that is used to spread information. Network routes aren't involved in group membership.

Starting a multicast

- Node wanting to start a multicast generates a multicast id *mid*.
- It looks up *succ*(*mid*) - the node responsible for that key. This node is promoted and becomes the root of the multicast.
- If P wants to join the multicast, it executes *lookup*(*mid*). This will send a message, from P to *succ*(*mid*), that P is requesting to join the *mid* multicast.
- This request message is sent via routing. While routing, let's say the message comes to node Q.
  - If Q has never seen a join request for *mid* before,
    - It will become a forwarder for *mid*.
    - P becomes a child of Q, and Q forwards P's join request to root.
  - If Q has seen a join request for *mid* before,
    - This means Q will already be a forwarder for *mid*.
    - P will become a child of Q, but its join request doesn't need to be forwarded to root anymore, as Q is already a member of the multicast tree.
- P is a forwarder for *mid* by definition.
- Messaging is done by sending a multicast message to root via *lookup*(*mid*).After that,root sends the message along the multicast tree.

Lamport Clocks

- If event a happens before event b, it is denoted by $a \rightarrow b$
- If a and b occur in the same process, they occur in the order they were observed.

  If

  $$T(a) \rightarrow T(b)$$

  then

  $$a \rightarrow b$$

- If a is the event of sending a message, and b is the event of the message being received (by another process),then $a \rightarrow b$.
- If a and b occur in 2 processes that never exchange any messages, then

  $$a \rightarrow b$$

  and

  $$b \rightarrow a$$

  are both false. This basically means that nothing can be said, and nothing needs to be said, about when the events happened, or which one happened first.

Lamport Algorithm

- Each process keeps an internal clock, that is incremented between 2 consecutive events.
- Let's say P sends a message that is received by the receiving process Q.
  - The event of sending the message is a.
  - The event of the message being received is b
- The message must also include the time when it was sent, i.e, $T(a)$.
- When the message is received, Q will set its internal clock to:

  $$max(T(a) + d, current\_clock)$$

  Generally, d is set to 1.

Totally Ordered Multicasting

Totally ordered multicasting is when events must occur in the same order on *all* nodes.

For example, consider the following scenario.

- A bank has 2 branches, each with their own copy of the database. One branch is in Mumbai, another in Delhi.
- A particular customer has 1000 in his account.
- Two events occur (nearly) simultaneously in real time:
  1. Bank HQ at Mumbai issues a 1% increase in amounts of customers, by interest or because of any other reason.
  2. The customer in Delhi deposits 100 in his account.
- In Mumbai, $1 \rightarrow 2$.

  The customer's final bank balance = 1110
- In Delhi, $2 \rightarrow 1$.

  The customer's final bank balance = 1111

This is an inconsistency. The bank will be okay with either amount in the customer's account, but the amount on all nodes should ***match*** with each other.

How to solve this?

We solve this using totally ordered multicasting. The order of events must be the same on all nodes, even if the order is different than the real-world order of events. It should just be the same.

- Lamport Clocks are used for this.
- When a process receives a message, it's stored in a queue ordered by timestamp.
- The receiver multicasts an acknowledgement to all nodes.
- Since lamport clocks are being used, $T(ack) > T(msg)$.
- A message will only be delivered to the underlying application if it is at the head of the queue and has been acknowledged by *all* other nodes.

This ensures that all nodes have the same copy of the queue.

Replica Management

Replicas are created for servers, to reduce latency and to provide resilience. They need to managed efficiently and consistency needs to be maintained across all replicas.

Content Distribution

What data should we propagate?

- Propagate only notification of an update. Replica servers will request the actual data from the main server whenever they need it.
  - This is used when writes>reads.
  - Needs a properly implemented invalidation protocol.
- Transfer data from one copy to another.
  - Transfers can be aggregated.
  - Used when reads » writes.
- Propagate the update operation.
  - Don't transfer the data.
  - Tell each replica what update to perform.
  - Send only parameters for the updates.

Push vs Pull Semantics

Push

- Server Based.
- Updates are propagated to replicas without request.
- Server has to keep a list of client replicas as well as their caches.

Pull

- Client Based.
- Client has to explicitly request data.

Hybrid

This approach is based on leases.

- Server issues leases to clients.
- While the lease is active, the server will push updates.
- After the lease has expired, the client has to pull updates, or request a new lease from the server.

Expiration Times for the leases can be based on:

- **Age** - If an object hasn't changed in a long time, it probably won't change in the future. We can provide a longer lease for such an object.
- **Renewal Frequency Based** - If a client requests an object more frequently, the expiration time for that client, for that object, will be higher.
- **State Based** - If a server has high load, it will grant leases with shorter expiry time.

Epidemic Protocols

- Used to implement eventual consistency.
- Propagate updates to all nodes in as few messages as possible.

Replicas can be of 3 types in this protocol.

- **Inactive** - Holds the update, and is willing and able to spread it.
- **Susceptible** - Hasn't received the update yet.
- **Removed** - Holds the update, but can't (or won't) spread it.

Anti-Entropy

Server P picks another server Q at random. They exchange info using one of the following approaches:

- P pulls from Q.
- P pushes to Q.
- P and Q pull and push from each other.

Pull-based approaches are generally better than push-based. This is because a susceptible node may not find any node that is willing to push to it. With pull-based approaches, it can pull the new info itself.

Combination of pull and push has been found to work the best in practice.

Gossipping Protocol

- P receives update of item x.
- P pushes the update to Q.
- If Q already had the update, P becomes disinterested in spreading it further.
- Otherwise, P and Q will both gossip to other servers.

A mix of anti-entropy and gossipping is the best

Deletion of Data

Deletion of data is done through the use of *death certificates*. An item that has been deleted is given a death certificate, and these certificates are propagated the same way as normal data would be.

The accumulation of death certificates becomes an issue. If a lot of items are deleted everyday, then a lot of storage space will be taken by just the death certificates.

Hence,old death certificates are removed using expiration dates.This always runs the risk that there was a node that didn't receive the death certificate.

A special server can be set up that never removes any death certificate. IF a deleted item is seen again (after its death certificate has expired), this special server will see it as well. This special server will circulate the death certificate once again.

Consistency Protocols

Consistency Protocols have the following classification:

- Primary Based
  - Remote Write
  - Local Write
    - Single Copy
    - Multiple Copies
- Replicated Write
  - Active Replication
  - Quorum Based

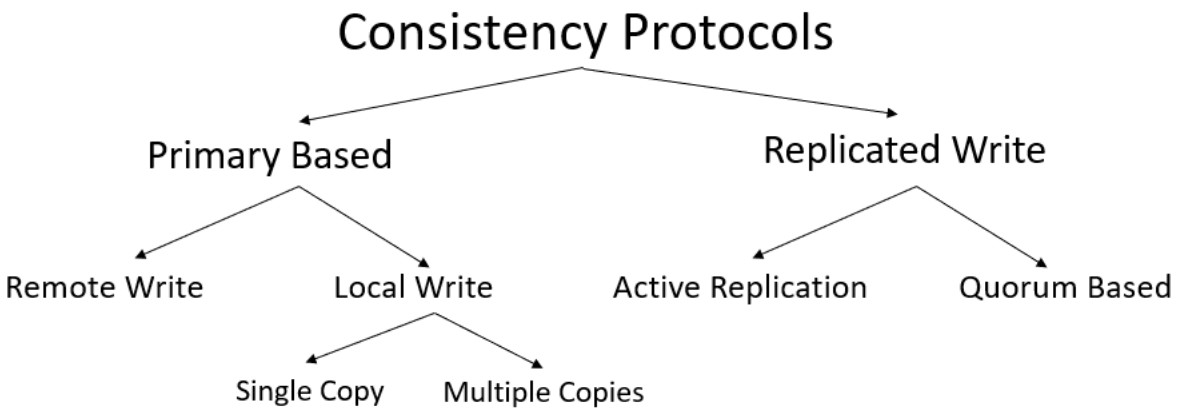

Figure 2: Consistency Protocols

Primary Based

Each data item has a **primary** node, which is the node responsible for it.

Remote Write

- A process wanting to write to a data item x locates x's primary.
- The update/write is forwarded to the primary.
- The primary will update its local copy, and forward the update to the backup/replica servers,
- Replicas will perform the update and send an acknowledgement to the primary.
- When all replicas have sent acks, primary will send ack. to the initial process.

Local Write

The status of which node is primary for a particular data item x can change.

Single Copy

Only the primary has the data item x. Processes who need x need to request it from the primary (for read access). If a process wants to update x:

- It locates the primary and moves x to its own location.
- It becomes the new primary for x.

Multiple Copies

Multiple copies exist, but one is primary.

If a process wants to update x:

- It locates the primary and moves x to its own location. It becomes the new primary.
- It will perform the update, ask backups to update, and receive ack. from the backups.

Replicated Write

In this, writes can be carried out anywhere, not just at the primary.

Active Replication

A special process carries out updates at each replica.

A sequencer can be used that assigns a unique ID to each update. Each update is then propagated with this unique ID.

Quorums

Clients must request and acquire permission from multiple replicas before a read/write operation.

For e.g, get the majority vote before reading/writing.

All processes that vote yes will perform the update and have a newer version of the data.

When reading, majority must say yes and they must all have the same version number of the data. This ensures that the client doesn't request the data from a replica that has an older version of the data.

Gifford's Method

Let the number of votes required to read be N_r , and the number of votes to write be N_w. Let the total number of nodes be N.

Gifford's method states that the value of N_r and N_w should follow the following two rules.

1. $$N_r + N_w > N$$
2. $$N_w > N/2$$

The first rule prevents read-write conflicts.

The second rule prevents write-write conflicts.

A special case of Gifford's method is *Read One Write All*

In this, $N_r = 1$ **and** $N_w = N$

By ensuring that updates are written to all nodes, we can read any one of them and be sure that we have the latest version of the data.

Fault Tolerance

Fault Tolerance has the following criteria.

1. **Availability** - It measures the degree to which the system is up and ready for use at any instant.
2. **Reliability** - It measures the degree to which the system is ready to use for longer periods of time.
3. **Safety** - Nothing catastrophic should happen even if the system fails.
4. **Maintainability** - The degree to which the system can be brought back up in case it fails, and the underlying issue can be fixed.

Availability	Reliability
Measures instant readiness of system	Measures readiness of system across long periods of time
Measures the ability of a system to do its job if needed	Measures the ability of a system to perform its function for some interval, without failure.
Measures the (average) amount of time the service was down	Measures the frequency/probability of failure
A system that goes down one millisecond every hour has high availability but low reliability	A system that works perfectly usually, but crashes for 2 weeks every August is highly reliable but has only 96% availability
Availability is 1- (1/3600000), Reliability(measured as failure rate) is once every hour	Availability is 50/52, Reliability (measured as failure rate) is once every year

Types of Failure

1. **Transient** - One -time.
2. **Intermittent** - Repeating sometimes.
3. **Permanent** - Always there.

Failure Models

1. Crash (Fail-Silent)
2. Fail-stop
3. Fail-safe
4. Omission Failure
5. Timing failure
6. Response Failure
7. Arbitrary Failure

Crash

Server halts, but worked correctly until it halts. No response is seen from the server after it halts.

AKA **Fail-Silent**

Fail-Stop

Server halts, but worked correctly until it halts. Other servers are able to detect it has halted.

Fail-Safe

Server produces junk output, and other processes are able to recognize it as junk.

Omission Failure

Server fails to respond to incoming messages.

- Send Omission
- Receive Omission
- Channel Omission

Timing Failure

Server's response is outside the specified time interval.

Response Failure

Incorrect response.

- Value Failure - Wrong value
- State Transition Failure

  Server goes through the wrong flow of control.

Arbitrary Failure

Server sends arbitrary data at arbitrary times.

Security

Types of Cryptosystems

Symmetric

AKA conventional cryptography/shared-key systems/secret-key systems.

Sender and receiver share the same key, which is used both for encryption and decryption.

The shared key must be kept private. Anyone in possession of the key can read encrypted messages.

The notation $K_{a,b}$ is used to denote a secret-key shared by A and B.

Asymmetric

AKA Public-key cryptography.

The keys for encryption and decryption are different, but form a unique pair. The key for decryption can only decrypt the data encrypted with its pair key.

Key for encryption - K_E.

Key for decryption - K_D.

One of the keys is made public, and the other one kept private.

The notation K_A^+ is used to denote a public key belonging to A, and K_A^- denotes a private key belonging to A.

If Bob wants to send a message to Alice, he should encrypt it using Alice's public key. Since Alice is the only person who possessed the corresponding private key, only she can decrypt the message.

Hash

A hash function H takes a message m of arbitrary length, and produces a bit-string h having a fixed length.

$$h = H(m)$$

By the pigeonhole principle, many inputs can result in the same output.

A good hash function should have the following properties.

- Given H and m, h should be easy to compute.
- However, given H and h, m should be computationally infeasible to compute.

i.e, the hash function should be **REPEATABLE** and **IRREVERSIBLE (ONE-WAY)**.

RSA

An asymmetric encryption algorithm named after its inventors - Rivest, Shamir and Adleman.

Based on the fact that prime factorization of very large numbers is a difficult and time-consuming process.

Steps

1. Take 2 very large prime numbers - p and q.
2. Calculate
$$n = p * q$$
$$z = (p-1) * (q-1)$$
3. Choose d such that d is relatively prime to z.
4. Compute e such that

$$(e * d)\%z = 1$$

Now, the number d can be used for decryption, and e for encryption.

One of these is kept private, and the other is made public.

Usage

Let the message to be sent be m. Here, m is interpreted simply as a binary number.

1. Divide m into fixed length blocks, m_i, such that:

$$0 \leq m_i \leq n$$

Each m_i is also interpreted as a binary number.

2. The sender calculates

$$c_i = (m_i^e)\%n$$

   All such c_i are calculated and concatenated into a single variable c.

3. c is sent to the receiver.

4. The receiver calculates

$$y_i = (c_i^d)\%n$$

Based on the properties of modulus, and the way we have chosen e and d, we can easily see that $y_i = m_i \forall i$.

This way, the receiver is able to reconstruct the message.

Properties of RSA

- RSA is secure because no method exist to (efficiently) find prime factors of large numbers.
- RSA itself is also computationally expensive, around 100-1000x slower than DES.
- It's generally used to securely share session keys, and then those session keys are used in a (faster) encryption algorithm, such as AES or DES.

Securely sending messages (Secure Channels)

Securely sending messages has the following problems to solve.

- **Confidentiality**

  No one else other than the intended recipient should be able to read the message.

- **Integrity**

  The recipient should have a way to be sure that the contents of the message weren't tampered.

- **Authentication**

  Both parties should have a way to be confident that they are sending messages to the right person.

Example using Challenge-Response Protocol

Alice is denoted by A and Bob by B. Their shared key is denoted by $K_{A,B}$.

1. Alice sends her identity to Bob, indicating that she wants to communicate with him.
2. Bob sends a challenge R_B to Alice. This could be a random number.
3. Alice encrypts it using their shared key $K_{A,B}$ and sends the result to Bob.
4. Bob receives the encrypted message, and decrypts it to check whether it contains R_B. If it does, then he knows the person on the other end is Alice, because no one else could have encrypted it with $K_{A,B}$.

Bob has now verified Alice's identity. But Alice hasn't verified Bob's.

5. Alice sends a challenge R_A to Bob. Bob must encrypt it with $K_{A,B}$ and send it back to Alice. Alice will decrypt it and verify whether it contains R_A.

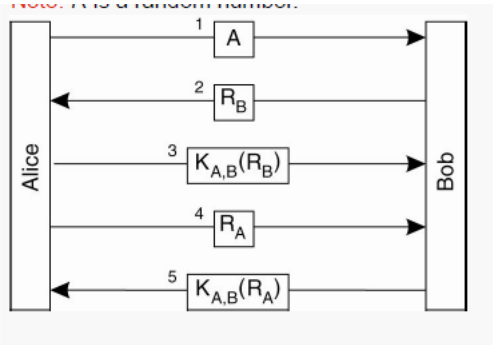

Figure 3: Challenge Response Protocol

Reflection Attack

Suppose we try to optimize the above approach.

Alice has to eventually send her challenge to Bob anyway, so she can just send her challenge when she's sending her identity (as part of step 1).

Similarly, Bob can return the response to Alice's challenge, and his own challenge, in a single message.

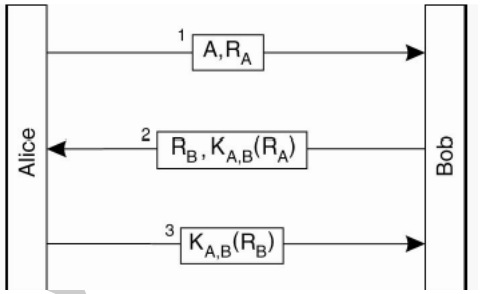

Figure 4: Optimization of Challenge Response

The above protocol can easily be defeated using a ***reflection attack***.

A reflection attack is a way to attack a challenge response system which uses the same protocol in both directions.

We basically try to trick the target into answering its own challenge.

Steps

1. The attacker initiates a connection to a target.
2. The target attempts to authenticate the attacker by sending it a challenge.
3. The attacker opens another connection to the target, and sends the target this challenge as its own.
4. The target responds to that challenge.
5. The attacker sends that response back to the target ("reflects" it) on the first connection.

Example using Reflection Attack

Chuck wants to pretend to be Alice, and talk to Bob. Chuck does not possess the key $K_{A,B}$. So, Chuck will trick Bob into encrypting his own challenge and giving it to Chuck.

1. Chuck sends a message to Bob ,containing Alice's identity and a challenge R_C. (Message 1)
2. Bob returns his challenge R_B and his answer to Chuck's challenge, $K_{A,B}(R_C)$. (Message 2)
3. Chuck needs to prove he is Alice, by encrypting R_B with $K_{A,B}$, and thus return the response $K_{A,B}(R_B)$ to Bob.
4. Chuck opens up a second channel to Bob, but this time he uses R_B as his challenge. He sends A and R_B in a single message to Bob. (Message 3)
5. Bob doesn't recognize that R_B is his own key, and encrypts it and sends $K_{A,B}(R_B)$ back to Chuck, with another challenge R_{B_2} (Message 4)
6. Chuck ignores the second channel, including the challenge R_{B_2}. He sends back $K_{A,B}(R_B)$ (received from Bob in the second channel), as a response in the first channel. (Message 5)

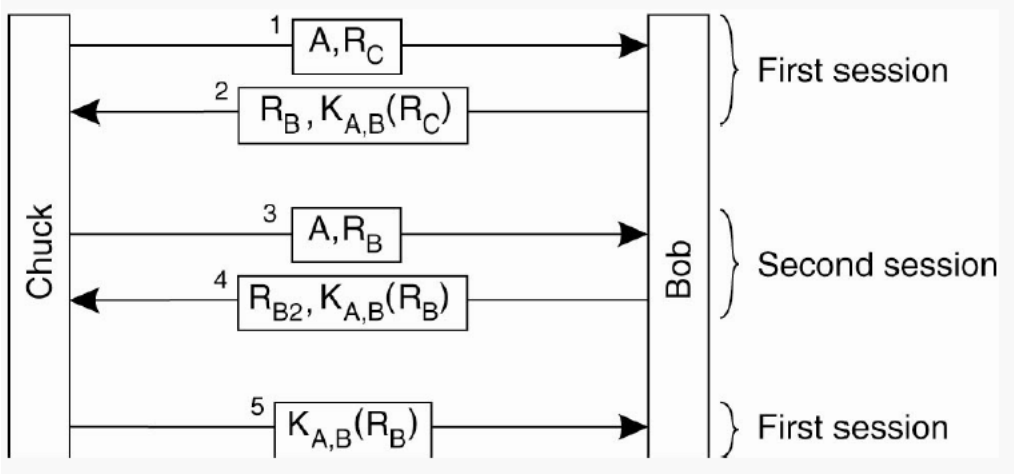

Figure 5: Reflection Attack

Mistakes Made

1. The same protocol was being used in both directions. It is always better to use a different challenge for the initiator and the responder.

   A basic example would be to make it so that challenges by the initiator (Alice/Chuck) are always odd numbers, and by the responder (Bob) are always even numbers.

   (However even this approach would be susceptible to a MITM (Man in the middle) attack)

2. Bob gave away valuable information in the form of the response $K_{A,B}(R_C)$ and $K_{A,B}(R_B)$, without knowing who he was giving it to.

   This wasn't violated in the original protocol, where Alice had to prove her identity first.

Mutual Authentication in Public-Key Cryptosystems

This system assumes there is some way to verify everyone's public keys.

i.e, if Alice wants to check whether a particular message was sent by Bob,

- Bob will encrypt it with his private key
- Alice can decrypt it using Bob's public key, and if the decrypted message is meaningful, she will know that Bob has sent it.

This assumes that Alice has a verified way, of getting Bob's public key, and being absolutely sure that the public key she has *indeed* belongs to Bob.

Generally, this involves the use of a trusted source, such as a KDC (Key Distribution Center), or a CA (Certificate Authority).

Example

Alice wants to set up a secure channel with Bob. Both possess each other's public key.

1. Alice sends a challenge R_A to Bob, encrypted with his public key K_B^+. She knows that only Bob will be able to decrypt this message.

   The message takes the form $K_B^+(A, R_A)$.

2. When Bob receives the message, he will do the following things:

   1. Decrypt Alice's message and extract R_A from it.
   2. Create his own challenge R_B.
   3. Generate a session key $K_{A,B}$ that can be used for further communication (using symmetric cryptography).
   4. Combine all the above 3 things into a single message and encrypt it with Alice's public key, and send it to Alice.

   The final message takes the form $K_A^+(R_A, R_B, K_{A,B})$

3. Alice encrypts Bob's challenge (R_B) with the session key $K_{A,B}$ and sends it back to Bob. This lets Bob know that the person on the other end is in fact Alice, since only Alice could have decrypted the previous message and extracted R_B and $K_{A,B}$ from it.

4. Alice and Bob communicate further using $K_{A,B}$ for this session. Once the session ends, $K_{A,B}$ is destroyed.

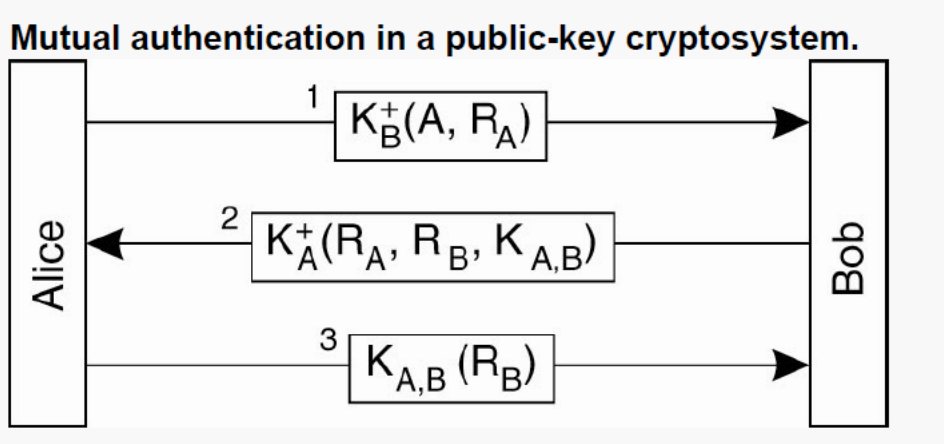

Figure 6: Mutual Authentication in a Public Key Cryptosystem

Digital Signatures

Confidentiality and Integrity needs to be maintained in secure channels.

- Alice needs to be sure that Bob cannot alter a message and claim that Alice sent it.
- Bob needs to be able to prove that a message indeed came from Alice, and that she cannot deny having sent it.

Digital Signatures are used for this. The document is signed using the sender's public key, which uniquely ties the sender to the message.

- Alice sends a message m to Bob. She encrypts it with *her* private key to create a ***signature***. The signature and the original message are sent to Bob.
  - If she wants to keep the message content a secret, she can encrypt the entire thing using Bob's public key.
  - The message will then be $K_B^+(m, K_A^-(m))$, where $K_A^-(m)$ is the signature.
- Message arrives at Bob.
  - If it's secret, he first decrypts it using his private key.

- He decrypts the signature using Alice's public key, and matches it with m. If the decrypted signature and m match, then he can be sure the message was sent from Alice and is untampered with.
- Alice cannot claim she never sent the message, or sent a different message, because Bob has the signed version of m, and only Alice could have signed it, since only she possesses her private key.
- Bob cannot claim Alice sent a modified message, because he would have to prove that Alice signed the modified message as well.

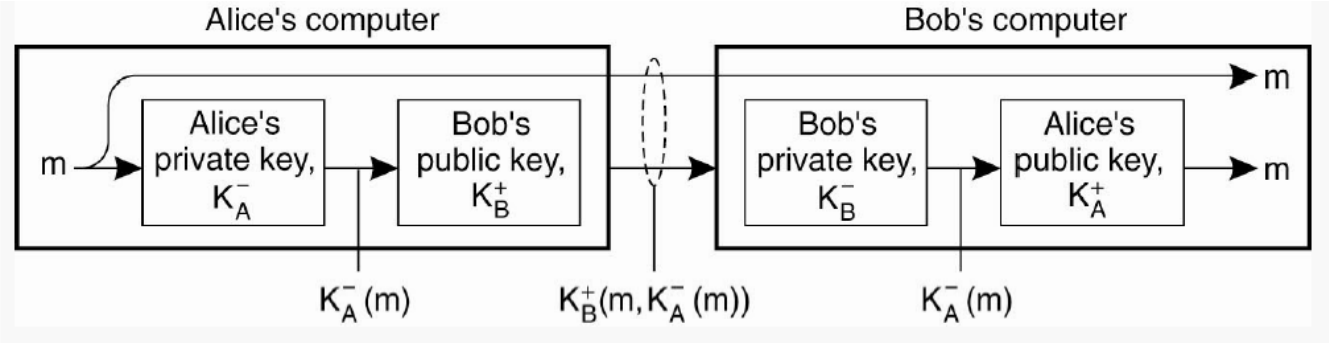

Figure 7: Digitally signing messsages

Issues with this scheme

- This remains valid only as long as Alice's private key remains private. If the key is stolen or leaked, Alice will have to generate a new key, and all messages signed using the previous key will then become worthless.
- If the message is long, encrypting the entire message may be computationally expensive.

A solution for the second problem is a **message digest**.

Message Digest

It's a fixed length string h that's computed from a message m of arbitrary length, using a hash function H.

If m is changed to m', then it's hash $H(m')$ will not be the same as before ($H(m)$). Thus, modifications will easily be detected.

Instead of signing m, Alice signs $H(m)$, which becomes the signature.

The message sent to Bob is now $K_B^+(m, K_A^-(H(m)))$, where $K_A^-(H(m))$ is the signature.

On Bob's end, Bob will hash the entire message himself, decrypt the signature, and compare the hashes. If they match, all is good.

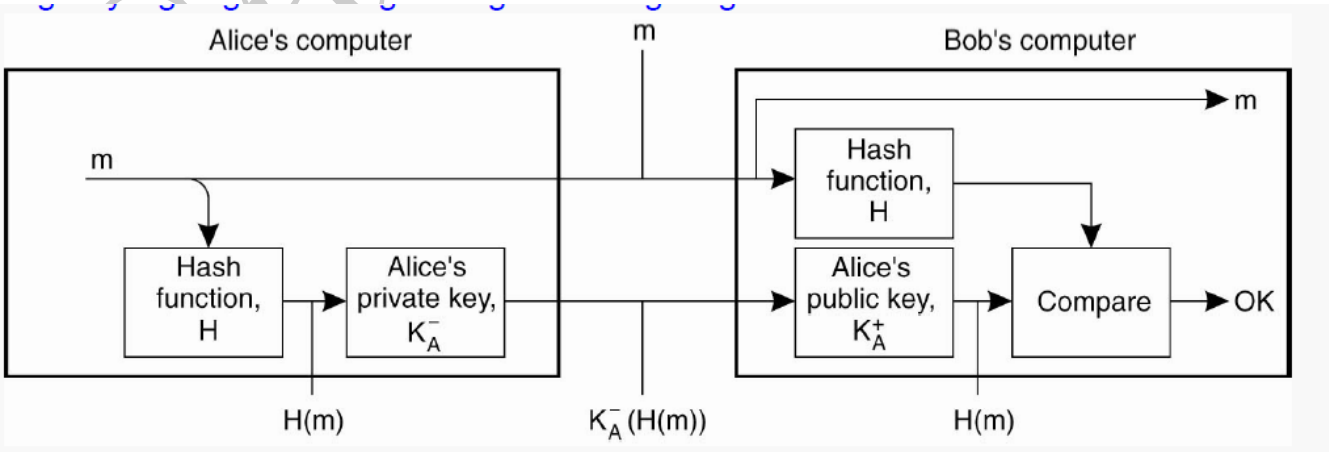

Figure 8: Digitally signing messages using digests

Diffie-Hellman Key Exchange

This is a method for 2 parties to exchange keys without the use of a third party.

1. Alice and Bob agree on 2 large numbers, n and g. Both numbers can be made public.
2. Alice chooses a large number x and keeps it private, and Bob chooses a large number y, and also keeps it private. Alice does not know y and Bob doesn't know x.
3. Alice sends $g^x \ mod \ n$ to Bob.
4. Bob sends $g^y \ mod \ n$ to Alice.
5. Alice computes $K_{A,B}$ as:
$$K_{A,B} = (g^y \ mod \ n)^x = g^{xy} \ mod \ n$$
6. Bob computes $K_{A,B}$ as:
$$K_{A,B} = (g^x \ mod \ n)^y = g^{xy} \ mod \ n$$

This way both Alice and Bob get the same session key, and no one listening from outside will be able to recreate it. This is based on the same principle as RSA.