Revision Distributed Systems

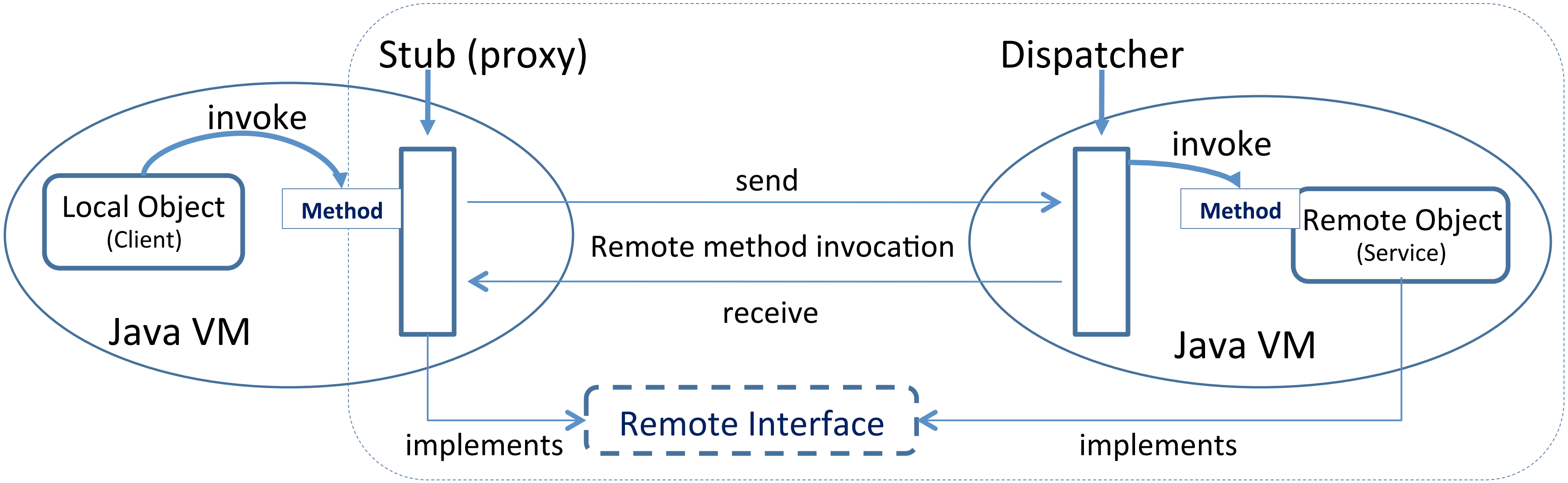
CS3524 Distributed Systems

Lecture 23

RMI

RMI Implementation

Distributed Object Model



- RMI middle-ware:
  - Stub
  - Dispatcher (uses Java reflection)

Serialization

- How to pass information between client and server?
  - So far, we have passed instances of java.lang.String from client to server
- How to pass more complex, programmer-defined data structures?
  - We need mechanisms to convert such data structures into and from byte streams – these byte streams can then be passed between client and server
- Java provides a simple means for object serialisation:
  - The interface **java.io.Serializable**
- If Java classes implement this interface, then objects instantiated from these classes can be translated into a byte stream that is sent across a network
- This byte stream is sufficient for the reconstruction of the serialized object when it is read by the receiving program

Callback

- Implementation in RMI:
  - We define a remote interface that contains a method for the server to call remotely and an implementation for it - these are the definitions for a “callback object”
  - Client instantiates, exports and registers a remote object that implements this interface – this is the “callback object”
  - Server provides a method in its remote interface that clients can use to inform the server of the remote object reference of their callback object – the server will record this reference
  - Whenever an event of interest for a client occurs, the server “calls the interested client” – it will call a remote method on the callback object

RMI Activation Service

- The ability to automatically generate remote objects, triggered by requests for references to these objects
- Robustness:
  - The distributed system is constructed in a way to handle and recover from disturbances and system failure, without loss of information, e.g. by
    - Restarting remote objects that were destroyed due to a system failure
- Scalability
  - A distributed system should be able to absorb high system load, increased/decreased service requests , e.g. by
    - Starting services/remote objects on demand

Threads

Threads

- A thread is a single sequential flow of control within a program
- Threads exist within a process (e.g. the Java Virtual Machine)
- Many threads may execute concurrently within such a process
- All threads share resources owned by this execution environment
- Multi-threaded execution is an essential feature of Java

Critical Sections

- Threads may execute concurrently and independently without interfering with other threads
- But: Threads (in most cases) interfere with each other – they access shared resources concurrently
  - A particular section of code specified in a Java class may be executed by multiple threads concurrently – we call this a ***critical section***
  - Critical sections of code have to be save-guarded in a special way – only one thread at a time should be able to execute such a section **in its completeness**
- Goal: **Mutual Exclusion** between threads – only one thread at a time executes a critical section in its completeness

Race Condition

- A ***race condition*** is an undesirable situation that occurs when two or more threads attempt to manipulate a shared resource concurrently
  - Two or more threads are able to access shared data and they try to make changes at the same time
- The result of such a computation depends on the sequence of how these threads are scheduled over time
  - A different schedule may result in a completely different interleaving of actions of the different threads and in different results
- "the threads are 'racing' against each other to win access to the shared resource"
  - E.g.: multiple read and write operations by different threads may result in one thread overwriting / removing a computation of another thread.

Monitor

- We need a mechanism to *coordinate and synchronise* thread activity. Java provides simple synchronisation mechanisms based on C. A. R. Hoare's widely-used *monitor* concept
- A monitor can be thought of as a *barrier* securing a *shared resource* with a **lock** :
  – If the resource is not used by another thread, a thread can *acquire* the lock and access the resource – the thread is regarded as "entering the monitor"
  – Other threads wanting to access the resource must wait (in a queue) until the lock is *released*

Critical Section in Java

- In Java, *critical sections* of code can be marked with the **`synchronized`** keyword to provide mutually exclusive access to an object
- The synchronized keyword indicates where a thread must acquire the object's lock
  - When a thread is scheduled for execution and enters a section of code that is enclosed with a **`synchronized`** construct, it has to acquire first a lock on this object
  - If another thread already holds this lock, our currently scheduled thread has to wait
- Please note: the whole object is locked for a particular thread!

Observer Pattern

- Class java.lang.Object provides the following methods (from the Observer Pattern):
  - wait()
  - notify()
  - notifyAll()
- These methods support an efficient transfer of control from one thread to other threads
- These methods can be used to handle the competition of threads for entering a monitor / shared object with a lock

Deadlock

- Necessary conditions for a deadlock to occur ("Coffman Conditions"):
  - Mutual exclusion:
    - At least one resource must be non-shareable – only one thread / process at a time may use this resource
  - "Hold and wait":
    - A thread / process is currently holding at least one non-shareable resource and is waiting for another non-shareable resource held by another thread / process
  - No preemption:
    - The operating system cannot simply free the resource, it must be given up by the holding thread / process voluntarily
  - Circular wait:
    - Two threads / processes: one thread waits for the release of a resource that is held by a second thread, which, in turn, waits for the release of a resource held by the first thread
    - N threads / processes: given n threads, the first threads waits for the second thread to release a research, the second waits for the third to release a resource , etc. , and the nth thread waits for the first thread to release a resource – circular wait.

Further Problems

- Livelock:
  - If thread A acts in response to the actions of thread B and thread B immediately reacts to thread A's action, they may be locked into an endless loop of mutually reacting to each others' actions, always performing the same set of actions
- Starvation:
  - can occur when threads can have different priorities
  - A thread is never scheduled for execution, because the scheduler always schedules threads with higher priority
  - Scheduling algorithm has to take this problem into account

Transactions

Transactions

- Transactions encapsulate a series of data manipulation operations (read/write)
- Transactions have a defined start where they are "opened"
- When a transaction is opened, all operations on data are regarded as occurring "preliminary"
- Transactions have a defined end with two possible outcomes
  - "**commit**": a successful termination of a transaction is regarded as a "commit" – all operations on data that occurred since the start of the transaction are *committed*
  - "**abort**": an unsuccessful termination of a transaction is regarded as an "abort" – all manipulation operations on data are *undone* or *rolled back*
- A transaction causes the system to move from one consistent state, at the application level, to another consistent state

The ACID Properties

- **Atomicity:** Either all of the transaction's operations are performed, or none of them
- **Consistency:** A transaction transforms the system from one consistent state into another
- **Isolation:** An incomplete transaction cannot reveal its intermediate state to other transactions until committed
- **Durability:** Once a transaction is committed, the system must guarantee that the results of the transaction will persist, even if the management system subsequently fails.

Serial Equivalence

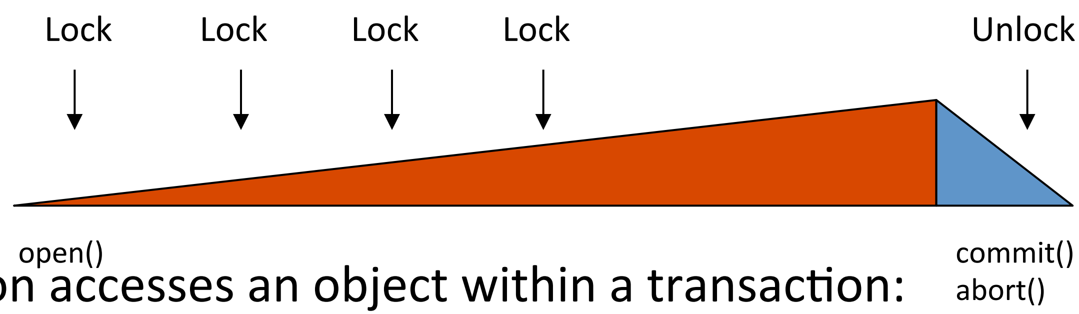
- Serial Equivalence
  - Two or more transactions produce the same result operating in an interleaved fashion, as if they would operate in a completely serialized fashion (each transaction is completed before the next one gains access to shared data objects)
- Serial Equivalence is used as a criterion for establishing concurrency control protocols
  - A transaction manager has to schedule operations of different transactions in a way so that interference and problems such as “lost updates” are avoided

Simple Locking

- If a transaction wants to read / write objects, it establishes locks for these objects
  - The server sets a lock on each object for a particular transaction before it is accessed (read or write operations)
  - The server removes these locks when the transaction is finished and not immediately after the operation – exclusive locking
- While an object is locked
  - only the transaction holding the lock can access/manipulate this object
  - Other transaction have to wait for the lock to be released or may be able to share a lock (this depends on the locking concepts used)
- The use of locks can lead to deadlocks !

Strict Two-Phase Locking

- No alternating lock /unlock during a transaction – transactions only set/ promote locks during their operation and release locks at commit / abort



- When an operation accesses an object within a transaction:
  a) If the object is not already locked, it is locked and the operation proceeds
  b) If the object has a conflicting lock obtained by some other process it waits until it is unlocked
  c) If the object has a non-conflicting lock set by another transaction, it shares the lock and the operation proceeds
  d) If the object has already been locked in the transaction, the lock will be promoted if necessary and the operation proceeds (if promotion is prevented by conflict, rule b) is used)
- When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction

Simple Exclusive Lock

- This simple strategy guarantees serialised transactions
  - The order of all conflicting pairs of operations is the same, because a conflicting transaction cannot proceed while it does not hold locks on objects it needs
- Important!
  - Strict two-phase locking – a transaction cannot set and release locks arbitrarily, there has to be (1) a phase of setting locks and then (2) a phase of releasing all locks, all locks are held until the release-phase at commit/abort
- In the example
  - Transaction U cannot proceed as long as Transaction T holds a lock for object b
- Problem
  - This is not the most efficient strategy; it unnecessarily restricts access to shared resources
- For example, two transactions that simply wish to perform a read() operation on an object would not interfere – allow shared read !

Shared *read* Locks

- Concurrency of transactions can be improved by distinguishing between read and write locks
  - Before a read() is performed, a ***read*** lock is set on an object
  - Before a write() is performed, a ***write*** lock is set
- Conflict rules
  - If transaction T has set a ***read*** lock then transaction U can set a ***read*** lock as well
  - If transaction T has already set a ***read*** lock then transaction U is **not** allowed to set a ***write*** lock until T commits / aborts
  - If transaction T has already set a ***write*** lock then transaction U is **not** allowed to set either ***read*** or ***write*** locks (***exclusive lock***)
- ***read*** locks are also called *shared* locks, as many transactions can set a read lock (or "share" it) on an object at the same time
- ***read*** locks guarantee that the object remains readable but no other transaction can set a ***write*** lock and make inconsistent updates

Lock Compatibility

For one object		*Lock requested for second object*	
		Read	***Write***
Lock already set	***None***	**OK**	**OK**
	Read	**OK**	**Wait**
	Write	**Wait**	**Wait**

- The fact that a lock has been set on an object by transaction 1 does not necessarily mean that transaction 2 cannot also obtain a lock
- The lock may be *shared* if they are both *read* locks
- Lock Promotion – make a lock more exclusive
  - A read lock can be "promoted" to a write lock if it is **not** shared by other transactions

Two-Version Locking

		Lock requested		
		read	write	Commit
Lock already set	none	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	---
	commit	wait	wait	---

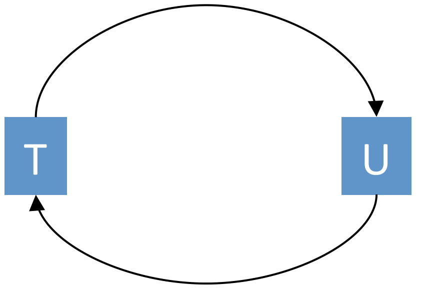
- Uses three locks:
  - Read, write, commit
  - A transaction can set a read lock if the object is unlocked or read locked
    - If the object has a commit lock, the transaction has to wait
  - A transaction can set a write lock, if the object is unlocked or read locked
    - If the object has a write or commit lock, the transaction has to wait
- At commit of a transaction, the transaction manager tries to convert the write locks into commit locks – if there are any outstanding read locks on the corresponding objects, the transaction has to wait

Deadlocks

- The use of locks may lead to deadlocks
- Definiton
  - A deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- The scheduler is responsible for detecting and breaking the deadlock
  - Deadlocks may be broken by simply aborting one of the transactions involved
- How do you choose which transaction to abort?
  - Abort the oldest
  - Abort depending on the complexity of the transactions
- Rather than detecting deadlock (an overhead), you could just use timeouts, but how long should the timeout be?

Wait-for Graph

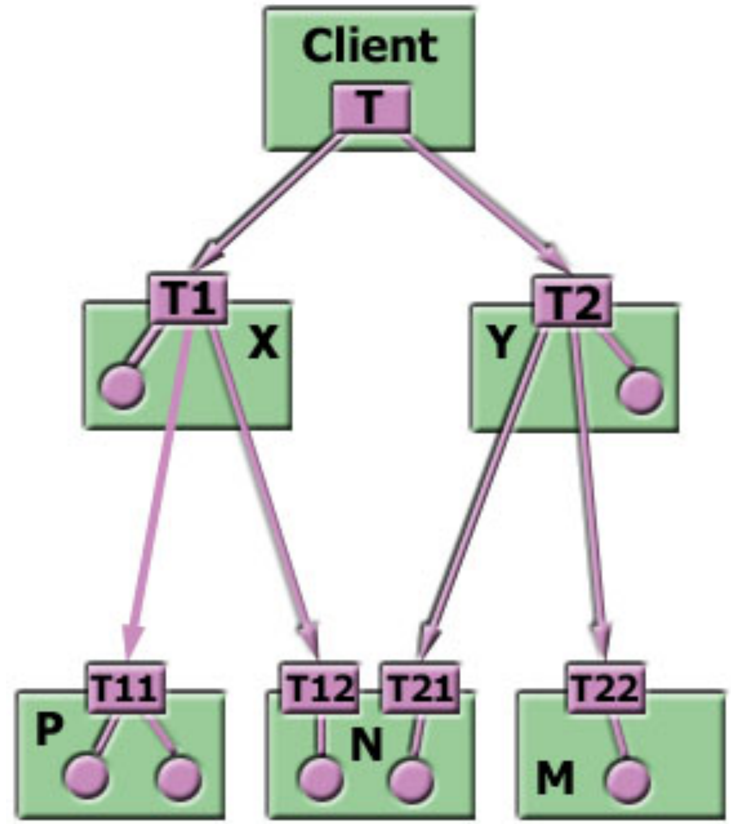
- Representation of a deadlock situation:
  - Nodes in this graph represent transactions
  - Edges between nodes represent wait-for relationships between current transactions
  - The dependency between transactions is indirect – via a dependency on objects
  - A cycle in the graph indicates a deadlock:



  - Transaction T and U wait for each other because of a lock on one object
  - None of these locks can ever be released
  - One of the transactions would have to be aborted to break this deadlock

Nested Transactions

- Nested Transactions have sub-transactons
- Sub-transactions can commit / abort independently
- A nested Transaction can only commit or abort if all its sub-transactions have completed
- When a sub-transaction completes, it makes an independent decision to provisionally commit or abort (abort is final)
- When a sub-transaction aborts, the parent transaction can decide whether to abort or not
- When the parent aborts, all sub transactions abort
- When the top level transaction commits, all transactions that committed provisionally, will commit finally



Committing Nested Transactions

- A transaction may commit or abort only after all its sub-transactions have completed
- When a sub-transaction completes, it makes an independent decision either to commit *provisionally* or to abort (abort is final)
- When a parent transaction aborts, all its sub-transactions are aborted, even those that have *provisionally* committed
- When a sub-transaction aborts, the parent can decide whether to abort or not
  - E.g.: this allows a parent to repeat a failed sub-transaction
- If the top-level transaction commits, then all of the sub-transactions that have provisionally committed can commit too, provided that none of their ancestors has aborted
- Note: the effect of sub-transactions are not permanent until the top-level transaction commits

Distributed Transactions

Atomic Commit Protocols

- Remember – Atomicity of transactions:
  - When a transaction comes to an end, either all of its operations appear to be carried out or none of them
- In case of a distributed transaction, these operations take place in more than one server
- How to complete a distributed transaction in an atomic fashion
  - Guarantee that either all of its operations (which are distributed across multiple servers) are carried out or none of them

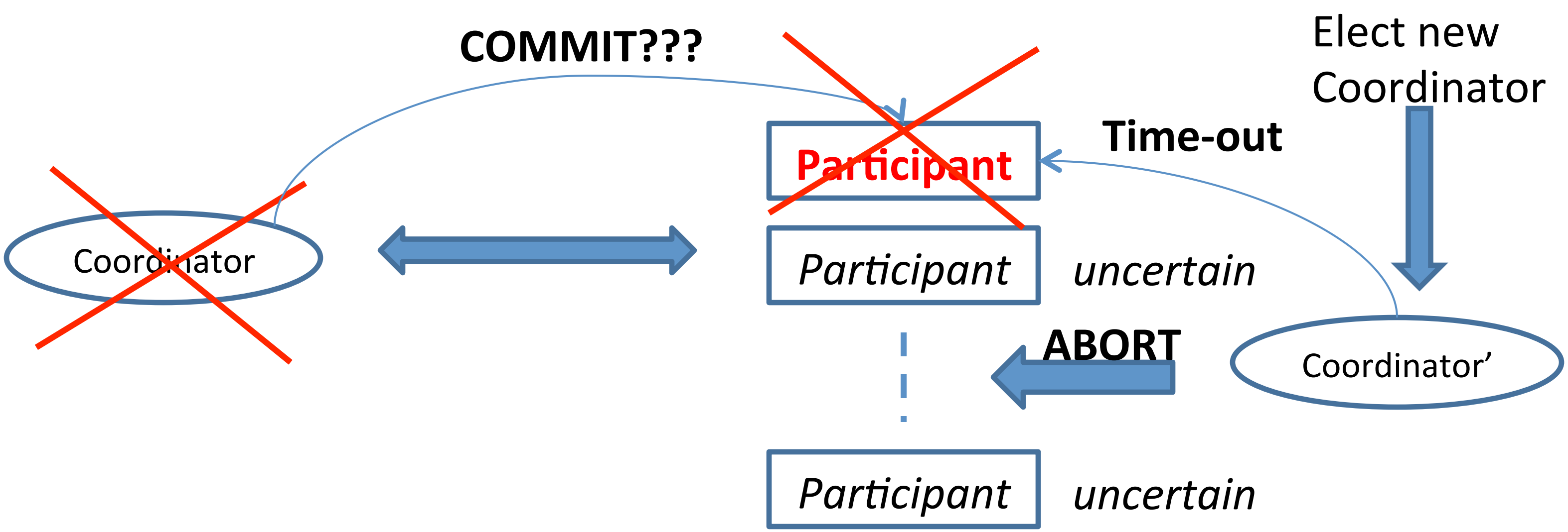
Two-phase Commit Protocol

- Is designed to allow any participant to abort the whole transaction – this requires informing all other participants and making sure that they get the message
  - Servers (coordinator, participants) can unilaterally decide to abort their part of a transaction
  - Due to the requirement of atomicity, if a part of a transaction is aborted, the complete transaction must be aborted
- Is based on a voting scheme: all the participants reach a consensus – either to commit or to abort
- Consists of two phases
  - Voting phase – ***prepare to commit***:
    - Coordinator asks participants whether they are ***prepared*** to commit
  - Completion phase - ***commit***:
    - Coordinator asks participants to ***commit***
- If a participant votes to commit, it must make sure that it can commit eventually, even if it crashes – participant must be able to recover from system failure by storing intermediate state

Coordinator Failures

- Serious problems occur in case of Coordinator failure
  - Participant received a canCommit? message in phase 1, is prepared to commit, but does not receive a doCommit message from the Coordinator
    - Participant is in an uncertain state and has to keep locks on data objects
  - Participants can wait for Coordinator to recover
  - Participants can elect a new Coordinator that restarts the 2PC protocol by sending new canCommit? Messages
- If coordinator fails in phase 2
  - It may have sent a doCommit or doAbort, before it crashed
  - One of the Participants may have received this information
  - Others can ask this Participant for the Coordinator's decision
  - If none of the Participants received a message from the failed coordinator
    - They can wait for Coordinator to recover
    - They can elect a new Coordinator that restarts the 2PC protocol by sending new canCommit? messages

Distributed Transactions



- Coordinator failed, one participant failed, no alive participant knows the coordinator's decision
- Problem
  - The coordinator's decision is unknown
  - It is unknown whether the crashed participant received a decision
  - The remaining participants may elect a new coordinator among them and vote – this new coordinator will time-out as the crashed participant does not deliver a vote and ***make an ABORT-decision***
  - **BUT:** If the crashed participant received a **COMMIT-decision**, it ***will commit data manipulations <u>after recovery</u>***
- **Possible CONTRADICTION !!**
- **All participants have to wait for the original coordinator to recover to receive its original decision**
  - **All manipulated data objects remain locked**

Coordinator fails in Phase 2

Scenario 3: Both Coordinator and one Participant fail

- Most critical case: both the coordinator and one participant failed
  - The failed Participant may have received a doCommit or doAbort, but which one? As it crashed, it cannot inform other participants
- If we assume that situation is resolved by electing a new coordinator / restarting the 2PC protocol:

• Participants elect new Coordinator • New Coordinator restarts 2PC and sends canCommit? in phase 1 • as one Participant failed, it will not receive the full set of votes • it will send a doAbort in phase 2 • the Participants alive will **abort**	• Failed Participant recovers – we assume it received a doCommit from the failed Coordinator • Recovered Participant **commits**

CONFLICT!!!

- New Coordinator cannot be elected !!
- Participants have to wait until original Coordinator recovers and sends doCommit or doAbort

Reliability and Availability

Replication

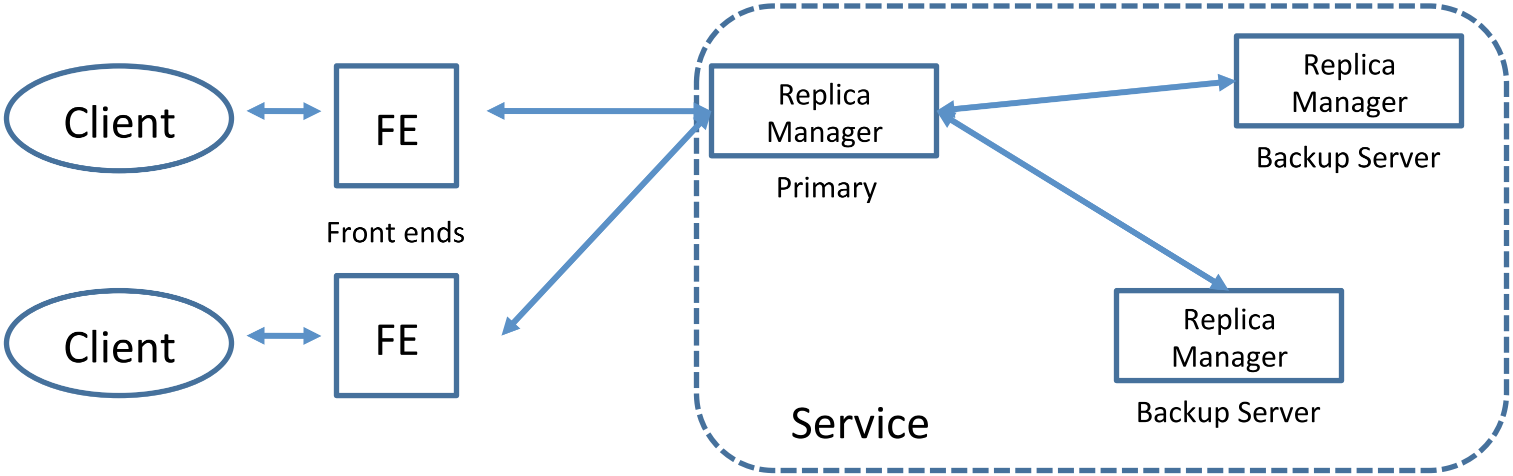
- Replication is the maintenance of copies of data at multiple computers
- Replication is a technique for enhancing services, it is key to the effectiveness of distributed systems, as it provides
  - enhanced performance,
  - high availability and
  - fault tolerance
- Example:
  - Caching of resources from web servers within browsers and web proxy servers
- Replication is of particular importance for mobile computing, because it operates in a more disconnected way
- We will consider two approaches:
  - Passive (primary-backup) replication
  - Active replication
- We will also consider transactions over replicated data

Replication

- Key Advantages
  - Performance Enhancement
    - Caching (replication) of data by a client may improve performance of data access
    - E.g. web browsers cache web sites, images and other data
  - Increased Availability
    - Availability is influenced by
      - Server failures
      - Network partitioning and disconnected operations – communication disconnection are due to user mobility and often unplanned
    - The proportion of time a server is available should be as close to 100% as possible
  - Fault Tolerance
    - Fault-tolerant systems and services guarantee ***strictly correct*** behaviour despite a certain number and type of faults

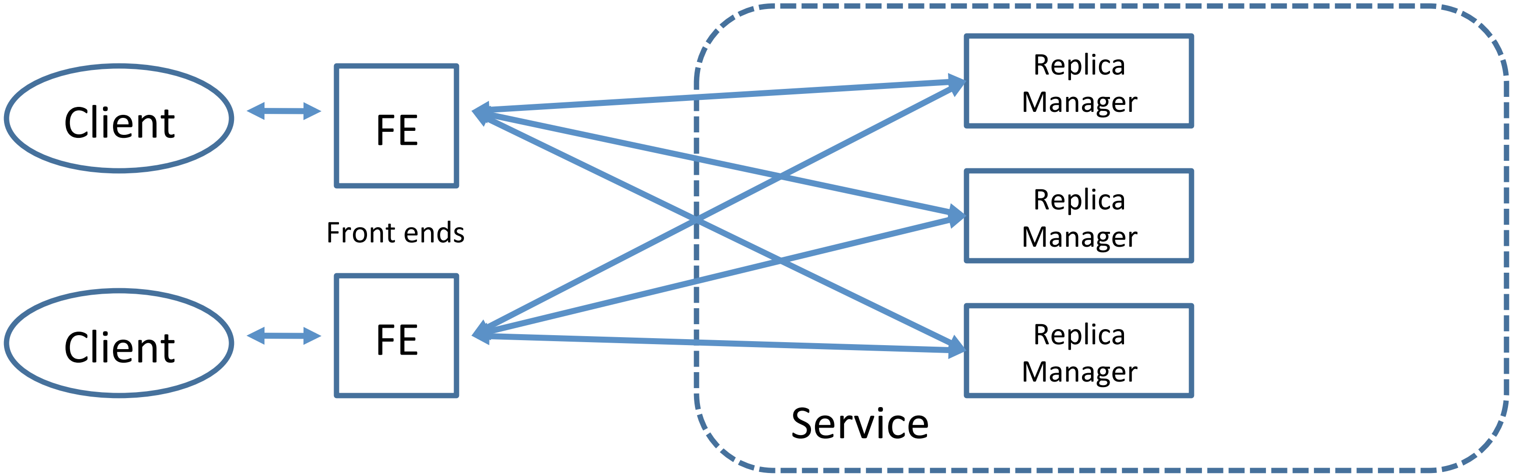
Passive Replication

"Primary-backup" Replication



- At any one time there is a single primary ("master") replica manager
- There are one or more secondary replica managers ("backups" or "slaves")
- Front-ends communicate only with the primary replica manage
- The primary manager performs operations on data and sends copies to the backups
- If the primary fails, one of the backups becomes the primary

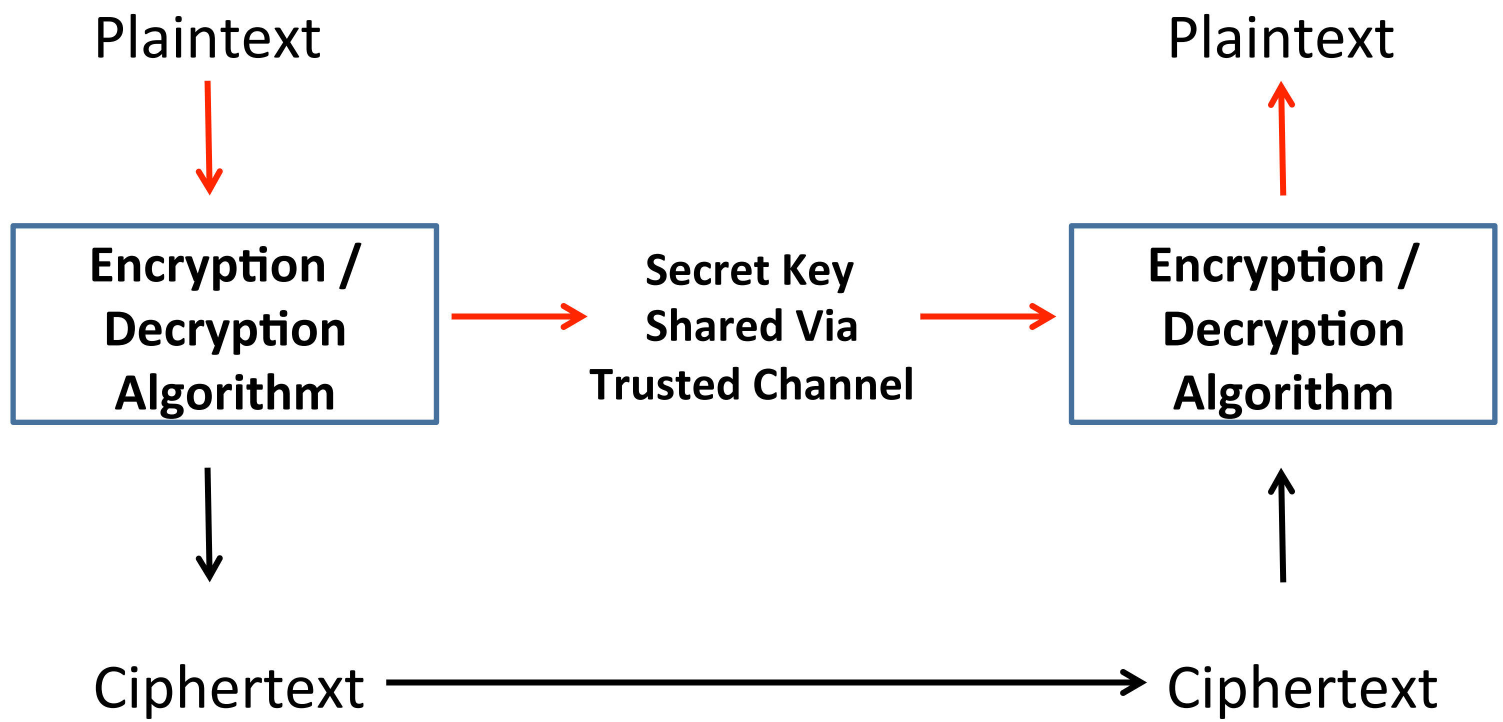
Active Replication



- All replica managers play the same role and are organised as group
- Client requests are multicast by the frontend to the replica manager group
- All replica managers process the request independently by identically
- All respond to the client request

Security

Symmetric-Key Encryption
(Secret Key Encryption)



- Same key is used by sender and receiver, hast to be shared via some trusted channel

Secret Key Cryptography

- Relies on one key for encryption and decryption
- Symmetric model of a cryptography system
- Encryption algorithm should be hard to break
  - Attackers should be unable to decrypt ciphertext or discover the key (even if they have a set of corresponding cipher / plain texts)
- The larger the key size, the harder to attack
- Sender and receiver must obtain copies of the secret key in a secure fashion
- As long as the key is kept secret, the cryptographic procedure does not have to be secret

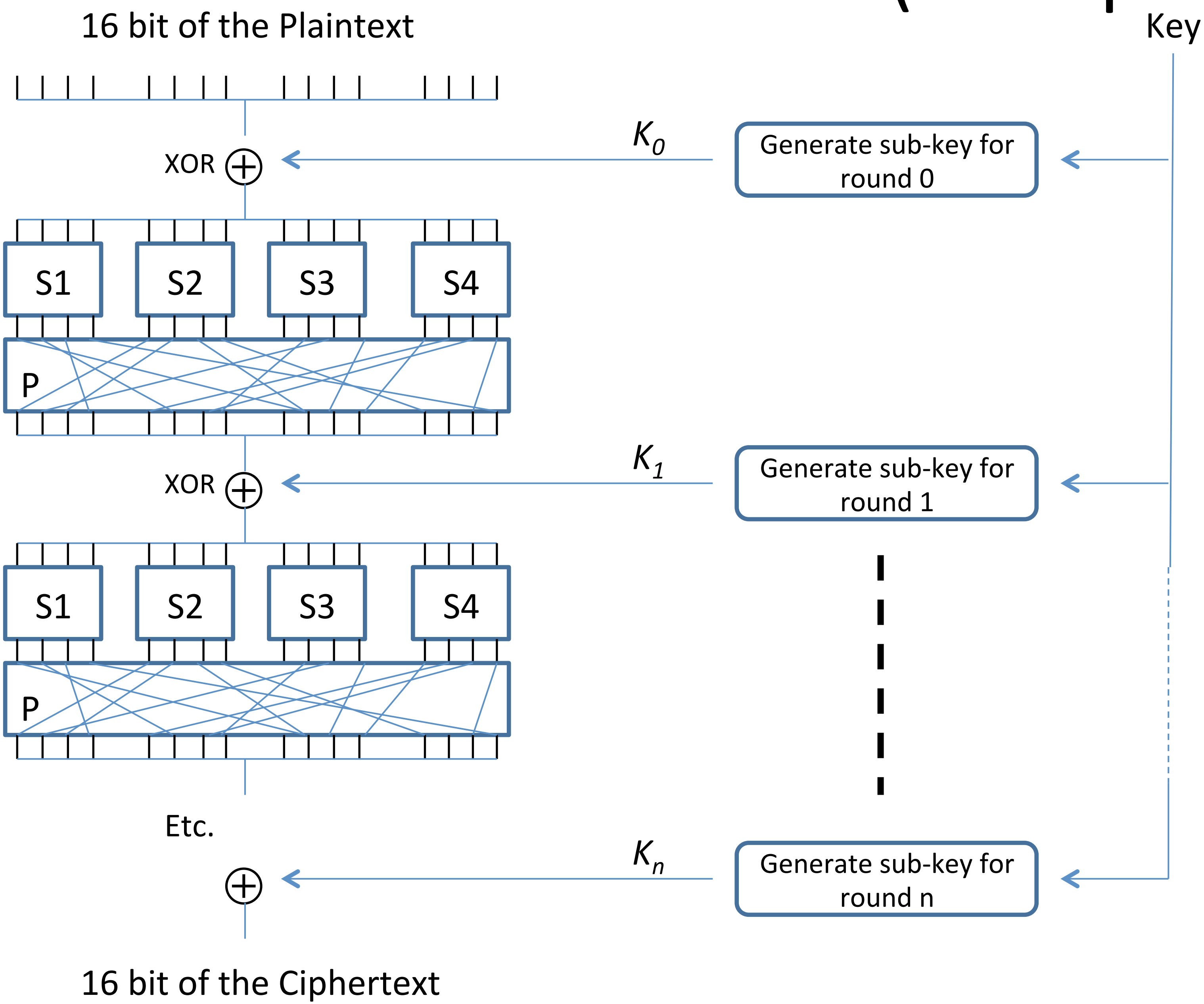
Block Ciphers

- Message is broken into blocks (usually 16 or 32bit words), each block is encrypted separately
- Operate with a fixed transformation procedure on large blocks of plaintext data
- Block Ciphers
  - Feistel Cipher
    - Is the first block cipher, which inspired subsequent cipher methods such as DES, etc.
  - Substitution-Permutation network

Substitution-Permutation Network

- SP-Networks (SPN) describe a series of substitution and permutation operations to be applied on plain text
  - The plaintext is separated into blocks (16bit words)
- The encoding operates over a sequence of rounds (“layers”), reapplying substitution and permutation operations over and over again to the output of a previous round

SP-Network (Example)



Substitution and Permutation

- Substitution in an SP-Network is performed with a so-called “substitution box” or S-box:
  - Are used to obscure or “confuse” the relationship between key and ciphertext
  - Takes as input *m* bits, produces a corresponding output of *n* bits
  - Implemented as a *m* x *n* lookup table
  - Substitution table is carefully designed to reduce vulnerability (Shannon: a change of one input bit should change at least half of all output bits)
- Permutation is performed with a so-called “permutation box” or P-box:
  - Takes the output of all substitution boxes as its input
  - Reorders (permutes) bits to produce output

Key Distribution

- With any symmetric algorithm, the key must be agreed upon by sender and receiver in a secure way
- Before 1976, key exchange was by far the biggest problem in secure communications
- Possible Strategies:
  - A key could be selected by A and physically delivered to B
  - A third party could select the key and physically deliver it to A and B
  - If A and B have previously used a key, one party could transmit the new key by encrypting it with the old key
  - If both A and B have an encrypted connection with a third party C, C could deliver a key on the encrypted links to A and B

Diffie-Hellman Key Exchange

- Developed in 1976, is a key exchange method where two parties exchange information that allows them to derive the same key, but never actually exchange the key
- Method:
  - Two parties, Alice and Bob, agree on a large prime number *p* and a small integer *g*; these two numbers are public
  - Alice picks a secret large random integer *a*, and calculates a number A:

$$A = g^a \bmod p$$

  - A becomes a public key, Alice transmits A to Bob
  - Bob picks a secret large random integer *b*, and calculates a number B:

$$B = g^b \bmod p$$

  - B becomes a public key, Bob transmits B to Alice
  - Alice computes the secret key: $K_{Alice} = B^a \bmod p$
  - Bob computes the secret key: $K_{Bob} = A^b \bmod p$
- Rules
  - p must be a prime number, ***p* > 2**
  - g must be a small integer, ***g* < *p***
  - a and b are large random integers, ***a* < *p*-1**, ***b* < *p*-1**

Diffie-Hellman Key Exchange

- Example

Alice	Bob
$p = 13$ $g = 6$	$p = 13$ $g = 6$
Random secret $a = 3$ Calculate public value $A = 6^3 \bmod 13 = 8$	Random secret $b = 10$ Calculate public value $B = 6^{10} \bmod 13 = 4$
Calculate secret key $K = 4^3 \bmod 13 = 12$	Calculate secret key $K = 8^{10} \bmod 13 = 12$

Public Key Cryptography

- Using one secret key poses a security risk
- A solution to this problem is “Public Key Cryptography”
- Two keys: public and private (secret) key
- The keys are matched so that
  - A message encrypted with the public key can be decrypted using the private key
  - A message encrypted with the private key can be decrypted using the public key

Applications for Public Key Cryptography

- Encryption / Decryption
  - The sender encrypts the message with the recipients public key
- Digital Signature
  - The sender "signs" a message with its private key, for this, a cryptographic algorithm is applied to the whole message or to a small block of data that is a function of the message (a "fingerprint" of the message, called a message "digest")
- Key Exchange
  - Exchange key information using the private key of one or both parties

RSA

- Encryption:
  - A ciphertext block *C* is the result of encryption of a plaintext block *M*, using the publicly known numbers *e* and *n*

$$C = M^e \bmod n$$

- Decryption
  - A plaintext block *M* is the result of decryption of a ciphertext block *C*, using the secret number *d*

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

RSA Key Generation

- To do:
  - Public Key: both sender and receiver must know the values of *n* and *e*
    - Calculate number *n* (maximum possible value of a plaintext / ciphertext block)
    - Calculate number *e* (a value needed for *e*ncryption)
  - Private Key
    - Calculate number *d* (a value needed for *d*ecryption)
    - only the receiver knows the value of *d*

RSA Key Generation

- Calculate n
  - Select two large prime numbers p and q, these are secret
  - Calculate: $n = p \times q$
- Calculate the public e
  - e is "relatively prime" to the Euler Totient $\phi(n)$
  - $e < \phi(n)$
- *What is*
  - "relatively prime" ?
  - Euler Totient $\phi(n)$?

RSA

- Relatively prime numbers:
  - Two integers n and m are relatively prime, if their greatest common divisor is 1: gcd(n,m) = 1
    - N and m do not share any common positive prime factors (divisors) except 1
- Euler Totient $\phi(n)$
  - Is the ***number*** of positive integers that are < n and that are relatively prime to n
    - E.g.: n = 10, {1,3,7,9} is the set of positive integers relative prime to 10, therefore: $\phi(n) = 4$
  - If n is the product of two prime numbers, p and q, then $\phi(n) = (p\text{-}1)(q\text{-}1)$
    - E.g.:
      - p = 3, q = 5, n = p x q = 3 x 5 = 15, therefore: ϕ(n) = (p-1)(q-1) = 2 x 4 = 8
      - n = 15, {1,2,4,6,7,8,11,13}

RSA Key Generation

- Calculate *n*
  - Calculate: $n = p \times q$, *p* and *q* are two large prime numbers
- Calculate the public *e*
  - We know:
    - If $n = p \times q$, *p* and q are prime numbers, then $\phi(n) = (p\text{-}1)(q\text{-}1)$
  - Choose *e*:
    - e is relatively prime to (p-1)(q-1) and $1 < e < (p-1)(q-1)$
- Calculate the private key *d*
  - $e \times d = 1 \bmod (p-1)(q-1)$
  - $d = e^{-1} \bmod (p-1)(q-1)$
- Result
  - Public key $K_{PUB} = \{e,n\}$
  - Private key $K_{PRIV} = \{d,n\}$

- *Encryption: encrypt a plaintext M to generate a ciphertext C via* $\mathbf{C = M^e \bmod n}$
- *Decryption: decrypt a ciphertext C to generate a plaintext M via* $\mathbf{M = C^d \bmod n}$

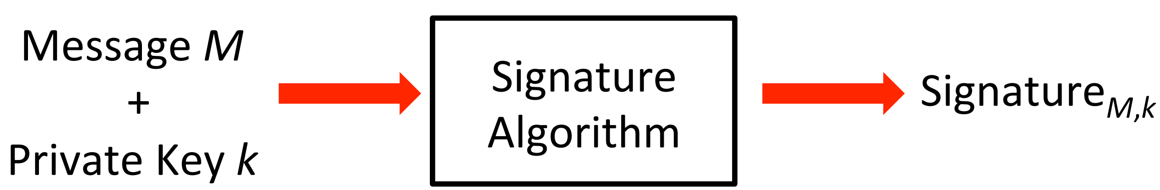
Digital Signatures

Digital Signatures

- Public key cryptography can also be used for creating digital signatures
  - Identifies reliably the originator of a send digital object (file / document, message etc.)
- Encryption of message with private key
  - Instead of a sender using the public key of the receiver, the sender's private key is used to create a unique signature

Digital Signatures
Creation from a Message

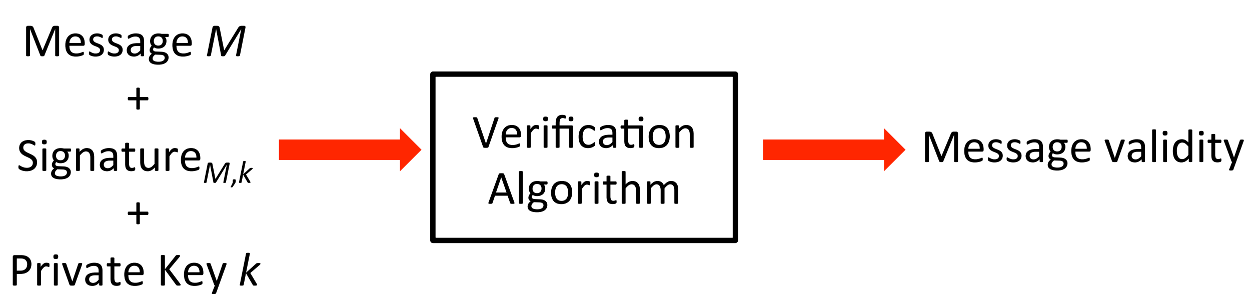
- A digital signature for a particular message is created by encoding a message with the private key
  - The message itself is not secret as anybody can decode it with the public key
- Only the person with the private key can produce the signature



Digital Signature

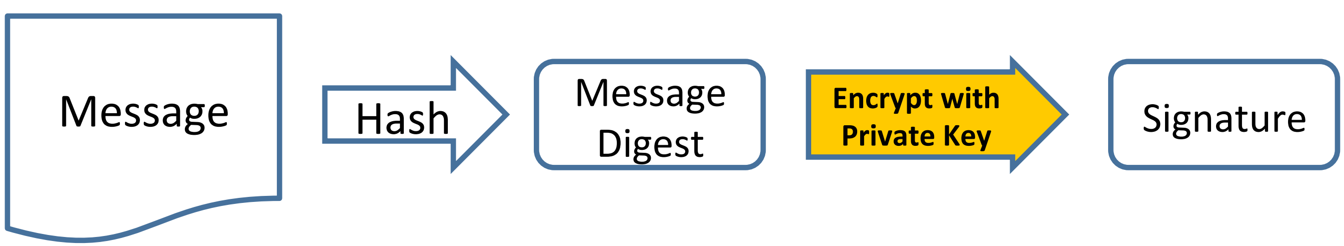
Verification

- A transmitted digital signature can be verified with the public key, identifies uniquely the sender.
- As public key is public, anyone can verify that the signature is valid



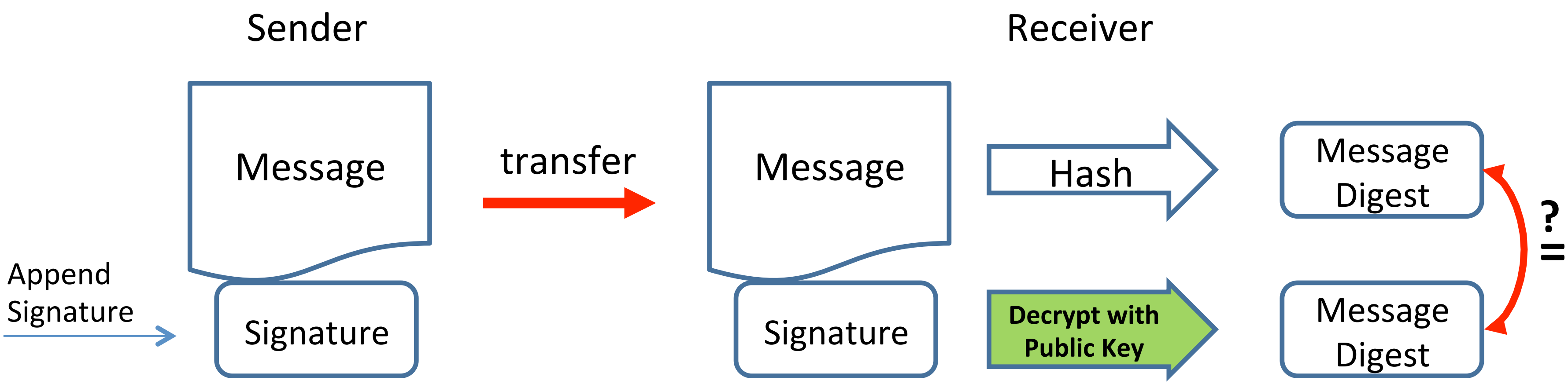
Digital Signature – Message Digest

- Encoding the whole message to produce a digital signature is not feasible
- Solution: create a “Message Digest”
  - Is a kind of “fingerprint” of a message, which is much shorter
  - the message digest is encrypted with the Private Key to create a unique signature for the message
- Calculation of the message digest
  - Apply a hash function to the message



Digital Signature – Message Digest

- Sender:
  - Creates signature from message digest
  - Appends signature to message and sends it
- Receiver
  - Recreates the message digest from the received message
  - Decrypts the signature with the public key of the sender
  - Compares the two results to check origin of message



Digital Certificate

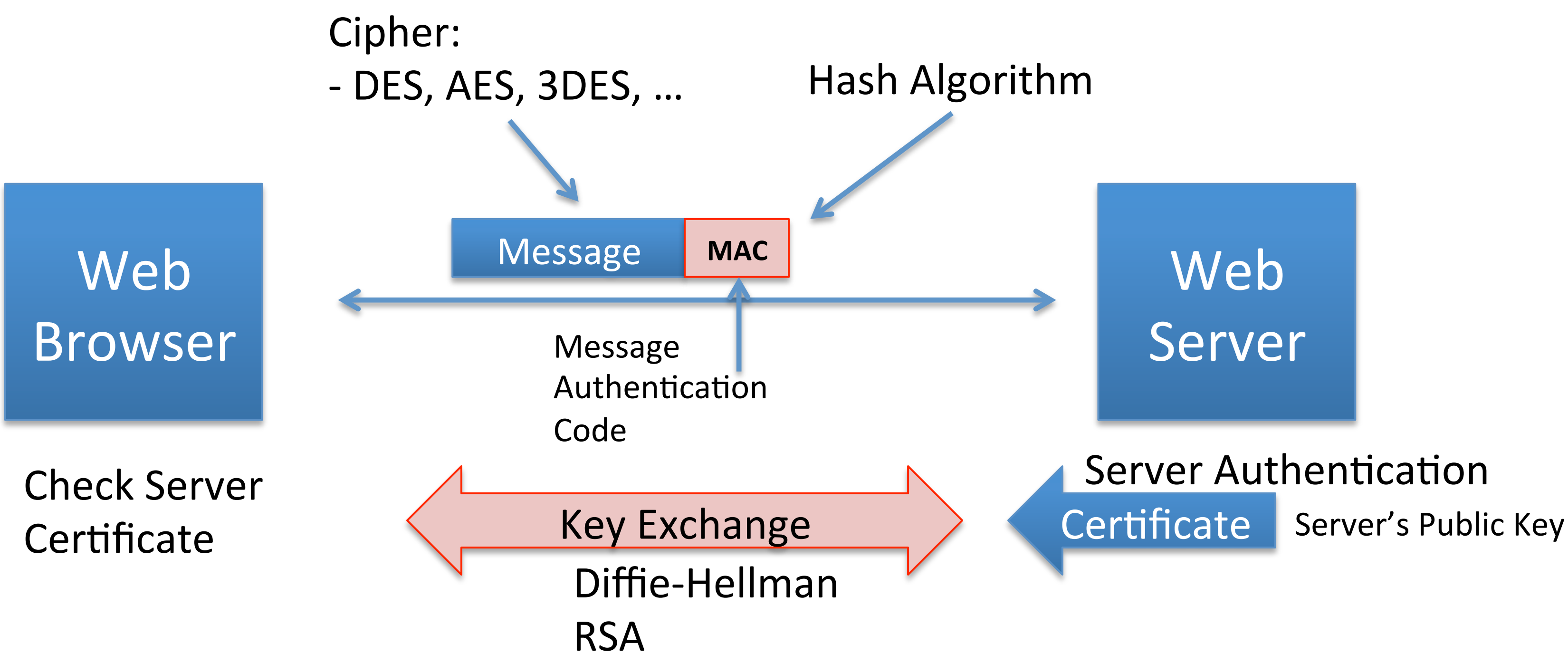
- Helps to address the problem of public-key distribution and authenticity
  - Certification:
    - digital certificates are introduced to proof authenticity of pubic keys
  - Validation:
    - ability to check that the binding of a public key to a certificate is authentic
- Is a "statement of originality" by a third party, the "Certificate Authority" (CA)
  - Binds a public key to a particular owner
  - A digital certificate itself is authenticated with a digital signature, signed by the CA with its private key
    - One's trust that a certified public key is original relies on one's trust in the validity of the CA's key

Public Key Infrastructure

- A Pubic Key Infrastructure provides the environment for the management of digital certificates
- Elements of a PKI
  - "Certificate Authority" (CA)
    - Is a trusted third party that issues and verifies digital certificates
    - Uses its own private key to sign digital certificats
  - "Registration Authority" (RA)
    - verifies the identity of users requesting information from the CA
  - A central repository for storing public keys

SSL Secure Socket Layer

SSL Secure Socket Layer



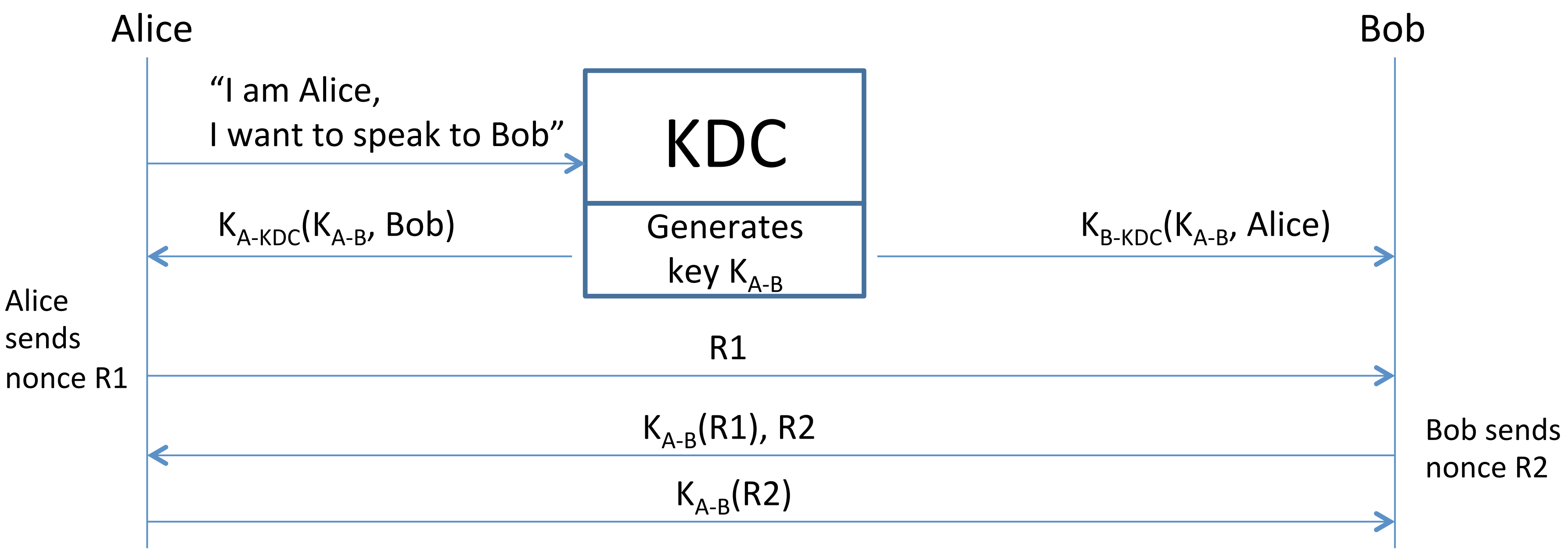
- Before encrypted communication
  - Negotiate cipher suite: crypto algorithm, hash algorithms for MAC
  - Authenticate Server (one-way authentication)
  - (maybe server also requests client authentication with certificate: two-way authentication)
  - Exchange information for secret key with Diffie-Hellman, RSA etc. key exchange

SSL Secure Socket Layer

- SSL can be viewed as a security layer that sits between the application layer and the transport layer
  - The client hands over data to SSL (e.g. An HTTP message for a web server), SSL then encrypts this data and writes it to a TCP socket
  - The server receives this encrypted data via its TCP socket, SSL takes this data, decrypts it and directs this data to the server for processing
- SSL uses symmetric key cryptography for encryption and decryption of data that is transferred
- To Do:
  - Verify that server is trustworthy (certificates)
  - Exchange a symmetric key between server and client

Mediated Authentication

- Goal: two parties eventually share a secret symmetric key ($K_{A\text{-}B}$) for communication
  - Communication partners have keys to contact the KDC ($K_{A\text{-}KDC}$, $K_{B\text{-}KDC}$)
  - Two nonces, "R1" and "R2" are used to proof authenticity



"Alice" has a key $K_{A\text{-}KDC}$ to communicate with KDC

"Bob" has a key $K_{B\text{-}KDC}$ to communicate with KDC

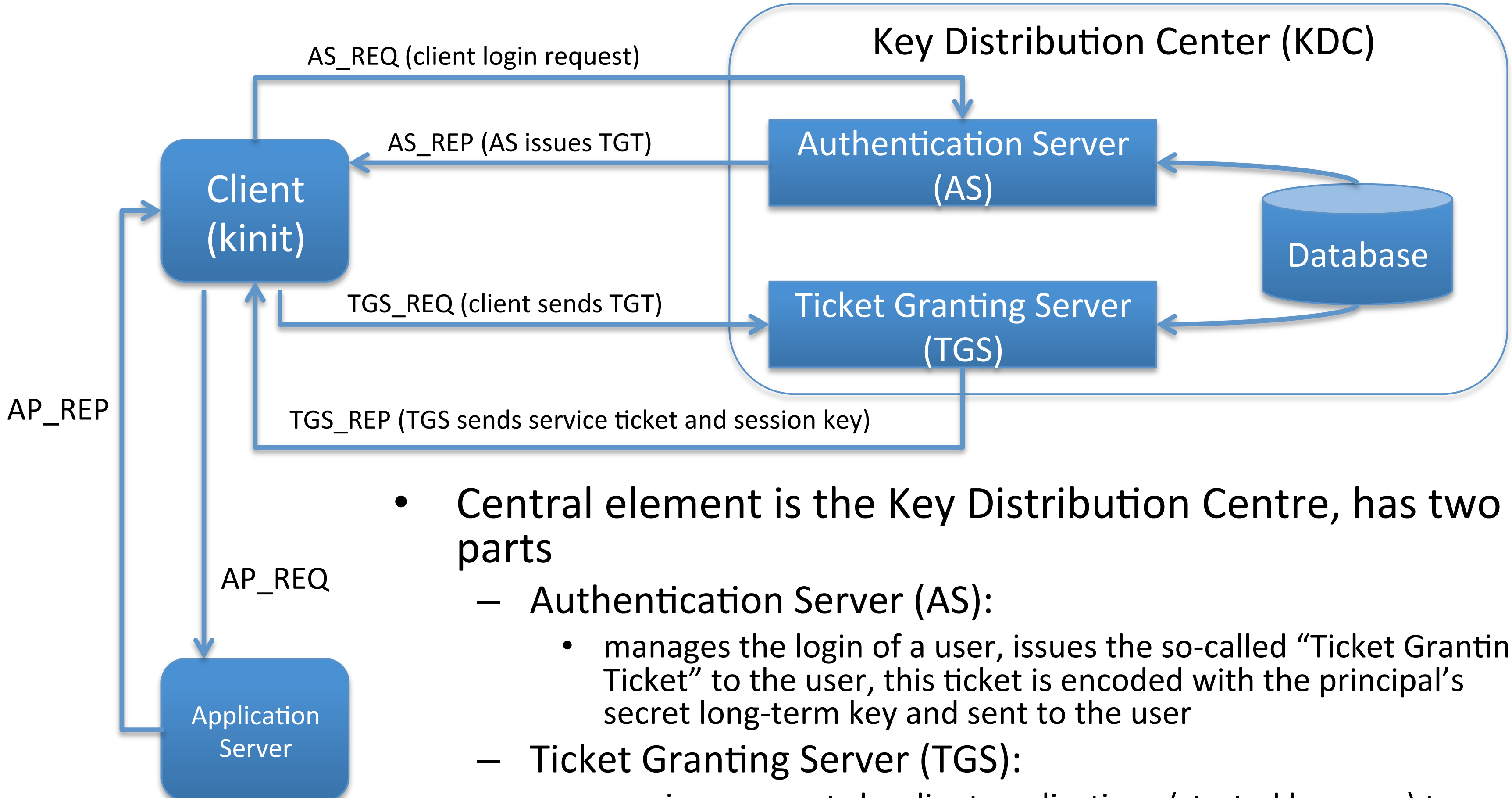
Kerberos

- Widely adopted and implemented in popular operating systems
  - See http://www.kerberos.org
- Kerberos uses time stamps as “nonces” in the mutual authentication phase of the protocol
- It uses mediated authentication with tickets

Kerberos Infrastructure

- Users and services are called “principals” – they have to be registered with a Kerberos domain, called a “realm” (managed by the Key Distribution Centre,
  - each principal is stored in a central database with an ID and a secret key
  - This secret key is a regarded the Principal’s “long-term” secret key
- Central element is the Key Distribution Centre, has two parts
  - Authentication Server (AS):
    - manages the login of a user, issues the so-called “Ticket Granting Ticket” to the user, this ticket is encoded with the principal’s secret long-term key and sent to the user
  - Ticket Granting Server (TGS):
    - receives requests by client applications (started by users) to access servers, client sends the TGT to the KDC for such a request
    - TGS calculates a secret session key for communication between client and server, is sent to client with a new service ticket (service ticket is encoded with the server’s long-term secret key)

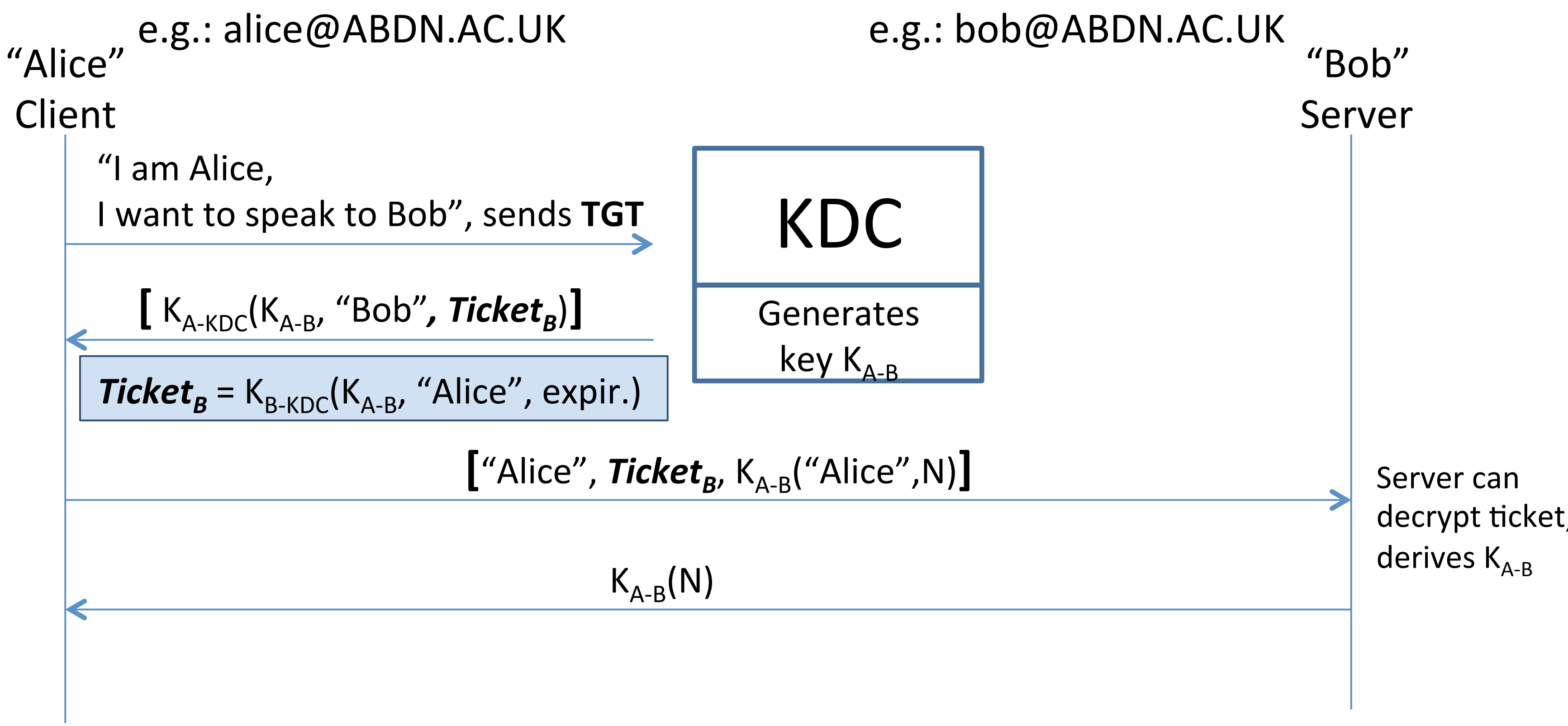
Kerberos Infrastructure



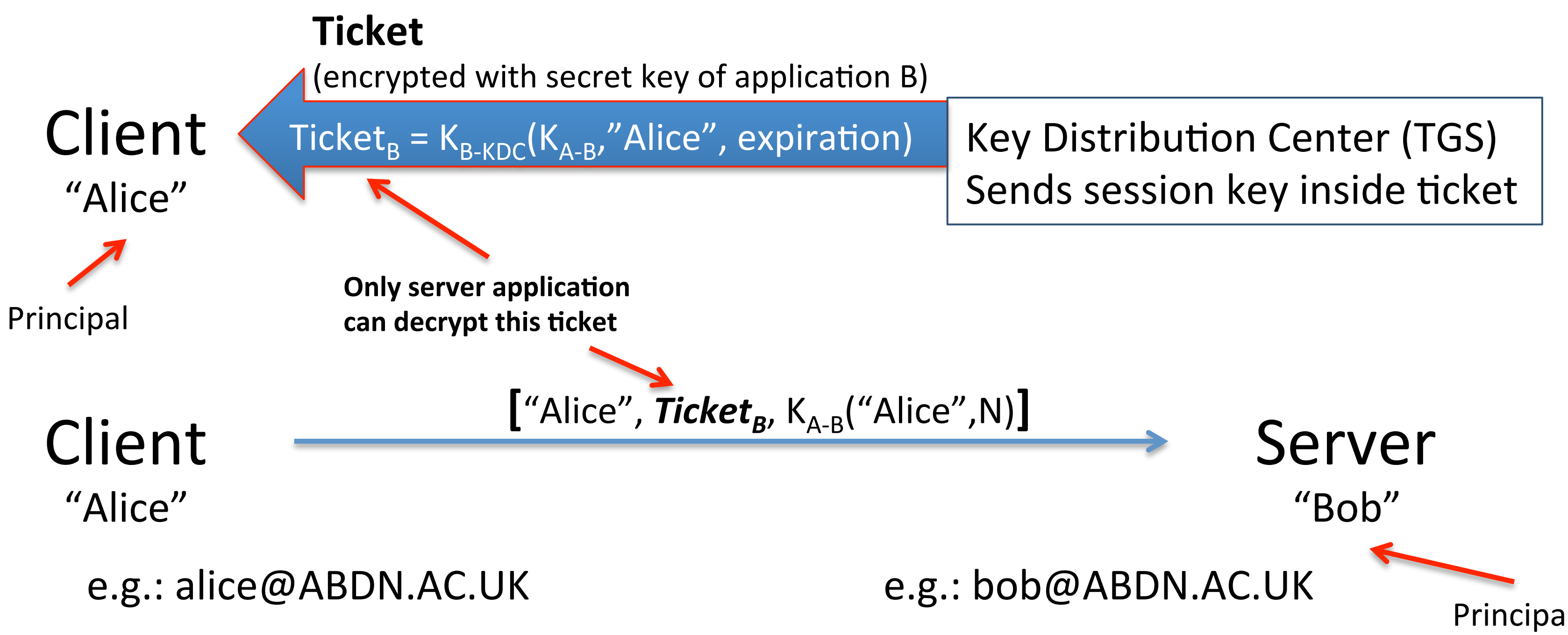
- Central element is the Key Distribution Centre, has two parts
  - Authentication Server (AS):
    - manages the login of a user, issues the so-called "Ticket Granting Ticket" to the user, this ticket is encoded with the principal's secret long-term key and sent to the user
  - Ticket Granting Server (TGS):
    - receives requests by client applications (started by users) to access servers, client sends the TGT to the KDC for such a request
    - TGS calculates a secret session key for communication between client and server, is sent to client with a new service ticket (service ticket is encoded with the server's long-term secret key)

Kerberos Authentication and Access to Applications

- Client sends TGT for authentication to KDC (TGS)
- KDC (TGS) responds with a session ticket encoded with the server's long-term key ($K_{B\text{-}KDC}$)
- A timestamp N is used as a nonce



Ticket Transfer



- Key Distribution Center transmits a ticket that is encrypted with the server's long-term secret key ($K_{B\text{-}KDC}$)
- Only server can decrypt this ticket, therefore authenticating the client

Kerberos Protocol

- Accessing Services
  - A client application (used by an authenticated user) sends user's TGT to the KDC, indicating that it wants to use a particular service
  - The KDC authenticates the client, checks access privileges to service, generates a random symmetric (short-term) session key $K_{A\text{-}B}$ for communication between client and server
  - The KDC sends a message back to the client, encoded with the shared key $K_{A\text{-}KDC}$: the value of $K_{A\text{-}B}$, and a **ticket** for accessing the service
    - $K_{A\text{-}KDC}(K_{A\text{-}B}$, $Ticket_B)$, where TicketB = $K_{B\text{-}KDC}$ ("client", $K_{A\text{-}B}$, expir.)
  - The client sends the ticket to the service, client also sends an **authenticator** for message to the service; the authenticator consists of the client name and a timestamp (nonce) *N* encrypted with $K_{A\text{-}B}$, that is $K_{A\text{-}B}$("client", *N*)
  - The service decrypts the ticket, using the secret key $K_{B\text{-}KDC}$, with that it will learn about the session key $K_{A\text{-}B}$
  - The service sends back the nonce to the client, encoded with $K_{A\text{-}B}$ to show that it received the secret session key and is "alive". This is the **mutual authenticator**.