

Cloud Applications

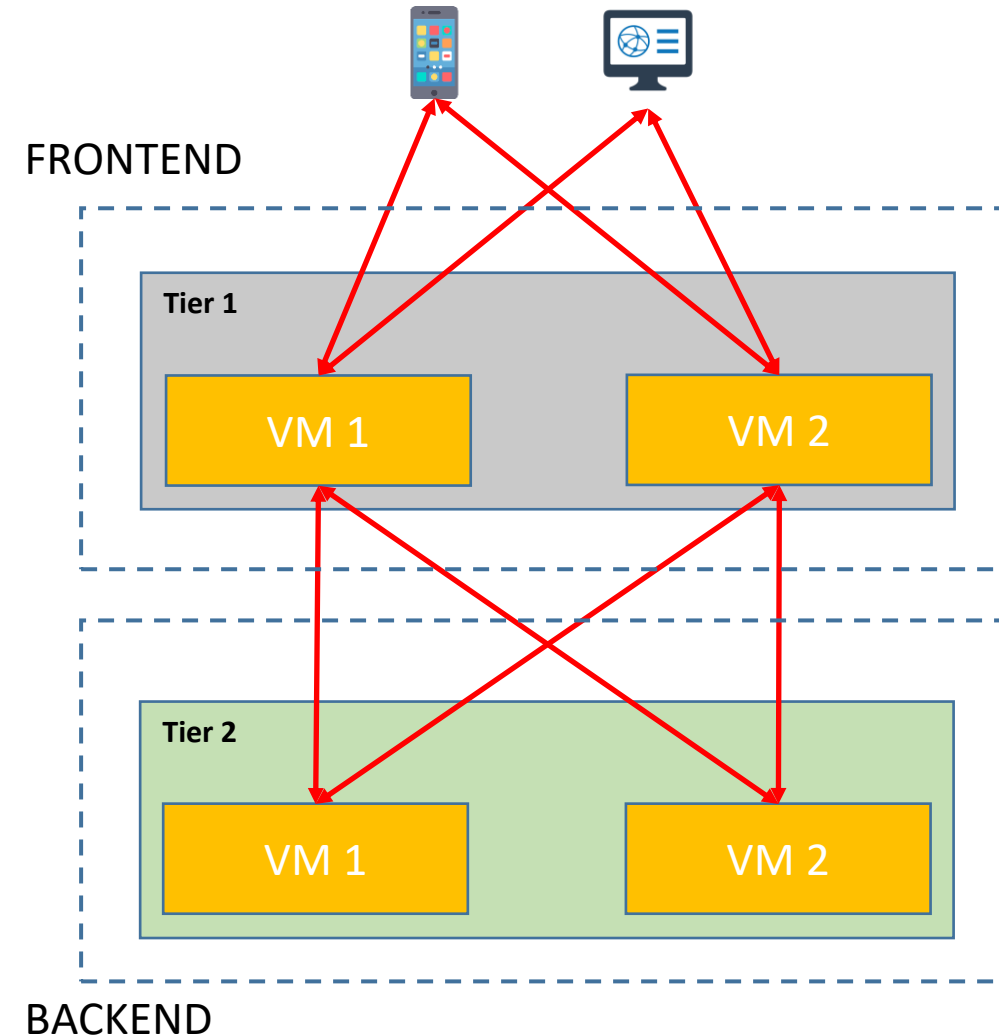
Scalable Cloud application design: Message oriented distributed systems and data replication

Reference:

- Slides
- Additional material: ***Distributed and cloud computing - Chapter 5.2 by Kai Hwang, Geoffrey C. Fox, Jack J. Dongarra***
- Additional material: ***Distributed Systems: concepts and design - Chapter 18 by George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair***

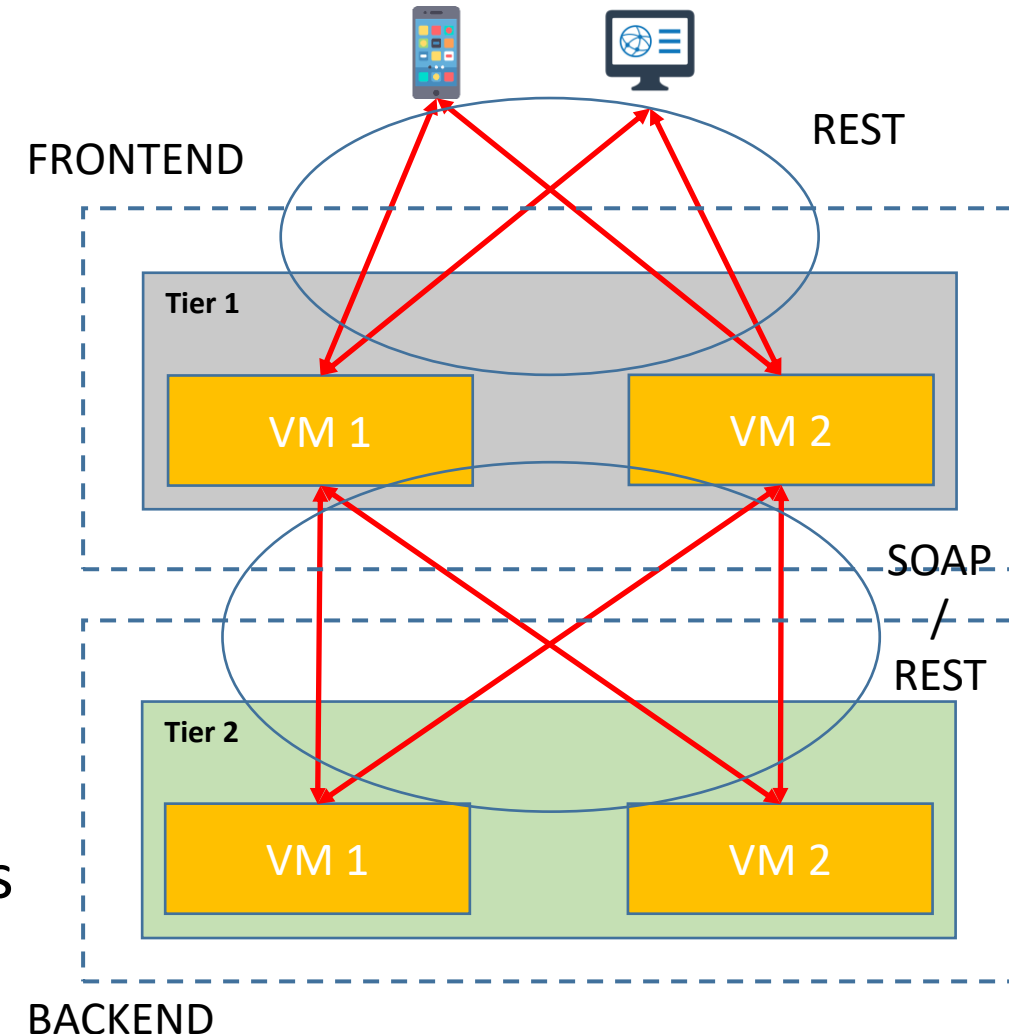
Recap: Cloud applications architecture

- Cloud applications are developed as distributed systems, whose implementation is the composition of different functionalities hosted on different VMs
- Such VMs are clustered into tiers, within each tier the VMs replicate the functionalities in order to ensure high availability and scalability
- Requests from clients are received by the VMs of the first tier, named frontend tier and then dispatched to the other tiers, named backend tiers



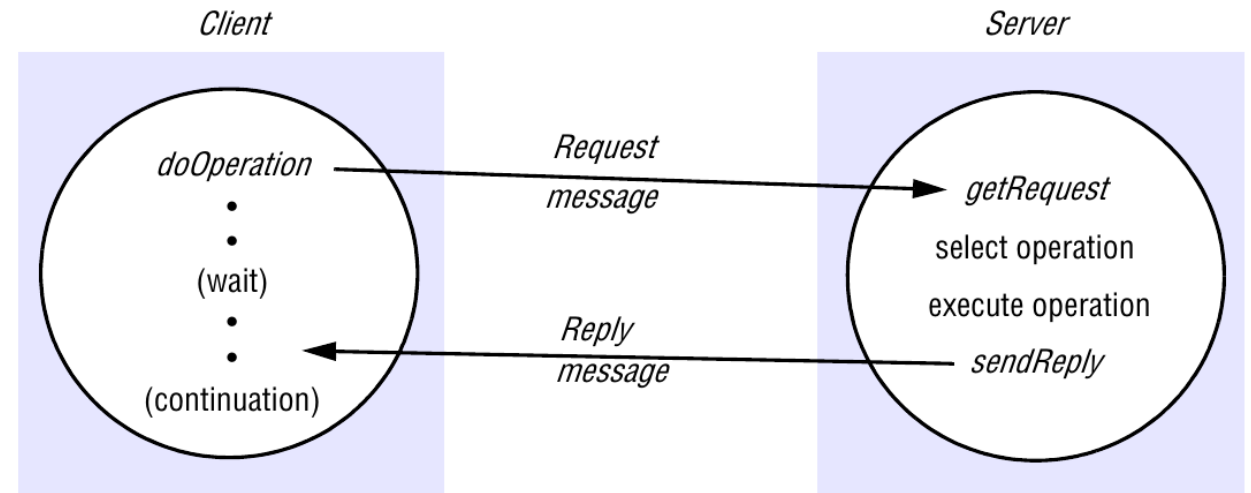
Recap: Design methodology

- A service oriented architecture is adopted: each VM hosts one (or more) self-contained software component, referred as service, that interact with others via **message passing**
- Two different solutions have been defined for service construction, i.e. to define the way messages are exchanged between software components:
 - SOAP
 - REST
- REST, due to its simplicity, is often adopted as interface towards clients, i.e. for the client/frontend communication
- SOAP adoption is decreasing recently, although it is still used in the frontend/backend communication and, rarely, in the client/frontend communication



Request-Reply communication

- Both SOAP and REST adopt a Request-Reply communication protocol
- *It implements a synchronous communication*: the client blocks until the reply arrives from the server (time-coupled)
- *It is reliable*: the reply is an acknowledgement of the reception of the request
- *It is a direct communication*: the client interacts directly with the server without intermediaries (space coupling), which introduces additional overhead and complexity



Space/Time coupling

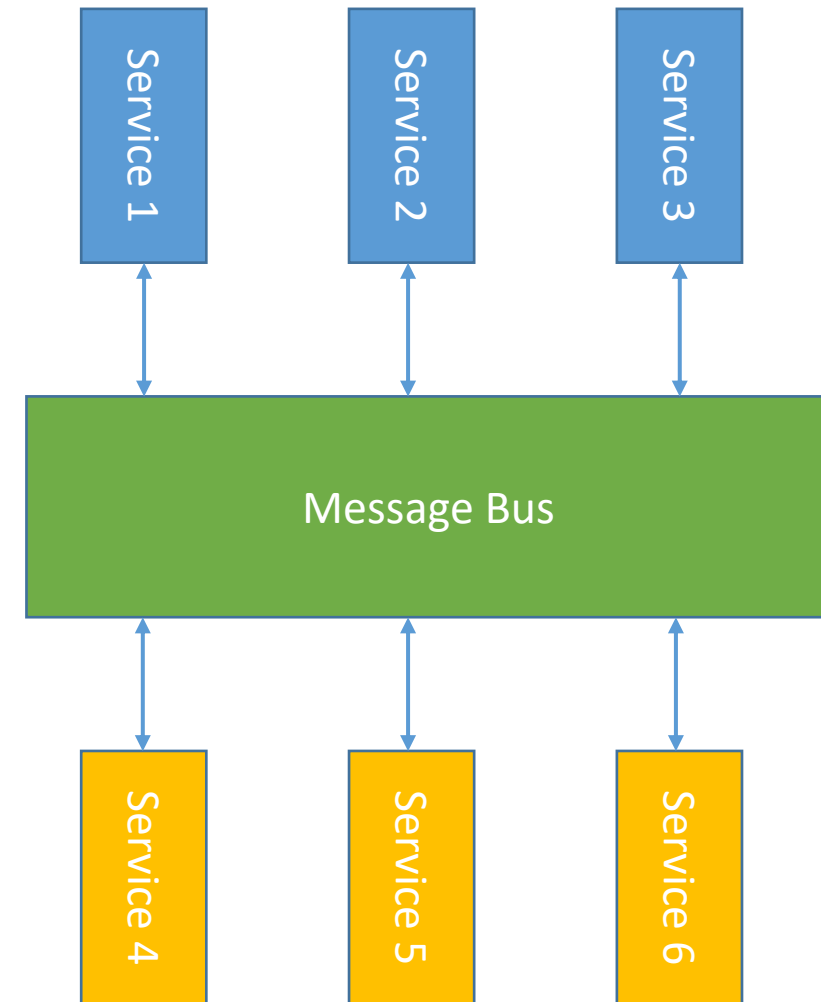
- Request-Response paradigm meets the requirements for the communication interface between clients and the cloud frontend: the paradigm is simple and reliable and client implementations can tolerate space and time coupling
- The communication between frontend and backend, however, is different: the backend is expected to have a varying number of VMs to support scalability through load-balancing or availability through data replication
- Supporting such dynamicity is paramount: the space and time coupling is not suited to ensure scalability in the communication inside the backend (or in the communication between the frontend and the backend)

Indirect communication

- An indirect communication paradigm is required in order to ensure:
 - Space uncoupling: the sender does not know or need to know the identity of the receiver(s), and vice versa. Senders/Receivers can be replaced, migrated, replicated at runtime
 - Time uncoupling: sender and receiver(s) have independent lifetimes. In a volatile environment the sender and receiver(s) do not need to exist at the same time to communicate (VMs can be destroyed/created dynamically)
- Space uncoupling is achieved by introducing an intermediary with no direct coupling between the sender and the receiver(s)
- Time uncoupling is ensured by the intermediary that adopts an asynchronous communication paradigm: the sender sends a message to the intermediary, then continues (without blocking) the intermediary takes care of delivering the message. The result is that there is no need to meet in time with the receiver to communicate

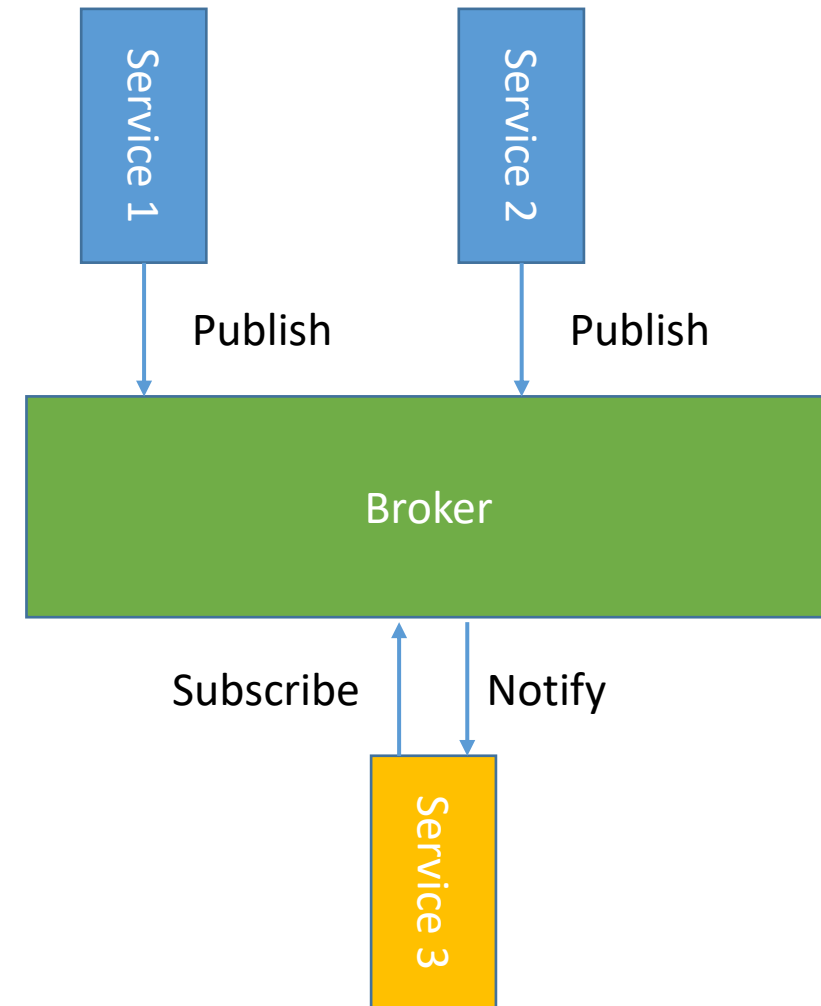
Message oriented communication

- Different indirect message-oriented communication paradigms have been defined for cloud applications
- In each one, source and destination does not open a direct communication channel, e.g. a TCP/UDP connection, but rather the sender injects the message into a bus
- The communication bus is implemented by an intermediary: the sender forwards the message to an intermediary, which takes care of forwarding the message to the receiver, and vice versa delivering the response to the sender, if any
- Different types of intermediary has been defined: broker or message queue



Message Broker

- A Publish-Subscribe system is a system where senders (named producers or **publishers**) publish structured data, usually an event, while receivers (named consumers or **subscribers**) express interest on a specific type of data through subscriptions
- At the core of the system, we have the *message broker*, which is responsible for relaying the data from publishers to subscribers, i.e. it matches subscriptions against published data and ensures the correct delivery of notifications
- The result is a one-to-many communication

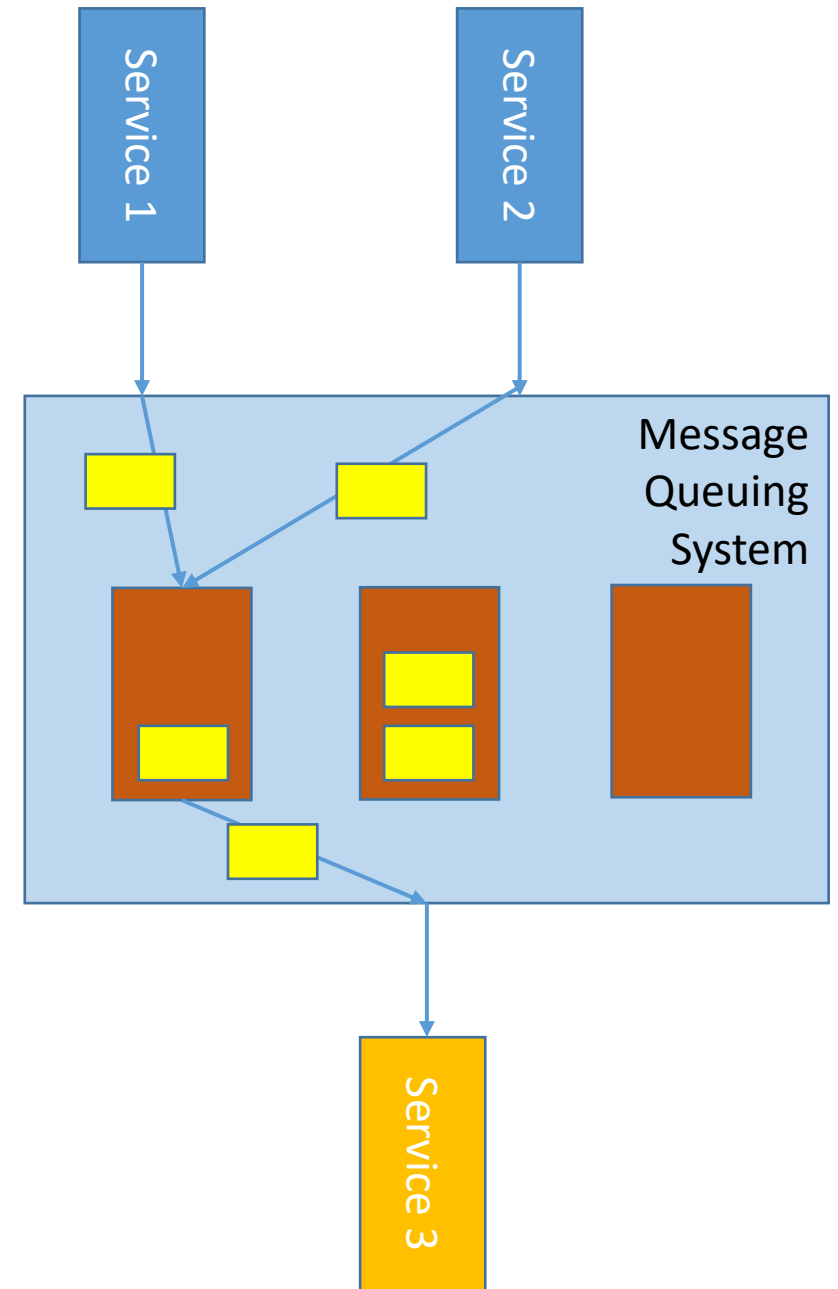


Subscription types

- The subscription model is general, different criteria can be used to filter define a filter for subscription:
 - *Channel-based*: publishers publish data to named channels and subscribers then subscribe to one of these named channels to receive all data sent to that channel
 - *Topic-based*: each data has a set of fields, one of them denotes the topic of the data. Subscriptions are then defined in terms of the topic of interest. This approach is equivalent to channel-based approaches, with the difference that topics are implicitly defined in the case of channels but explicitly declared as one of the fields in topic-based approaches
 - *Type-based*: subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter

Message Queue

- Message queues provide a point-to-point message exchange between the producer and the consumer: the sender places the message into a queue, and it is then removed by the receiver
- Messages queues are implemented by the Message Queueing System
- Inside the system, different queues can be instantiated: a producer can send messages to a specific queue while a consumer can receive message from a specific one



Message Queueing System Operations

- The set of operations offered by the message queueing system is simple:
 - Send, issued by the producer to place the message in the queue
 - Blocking Receive, issued by the consumer to block until an appropriate message is available
 - Non-blocking receive (polling operation), issued by the consumer to check the status of the queue, it will return a message if available or a not available indication otherwise
 - Notify, issued by the message queueing system to notify a consumer when a message is available in its associated queue (the consumer has to subscribe first updates on a certain queue)

Messages

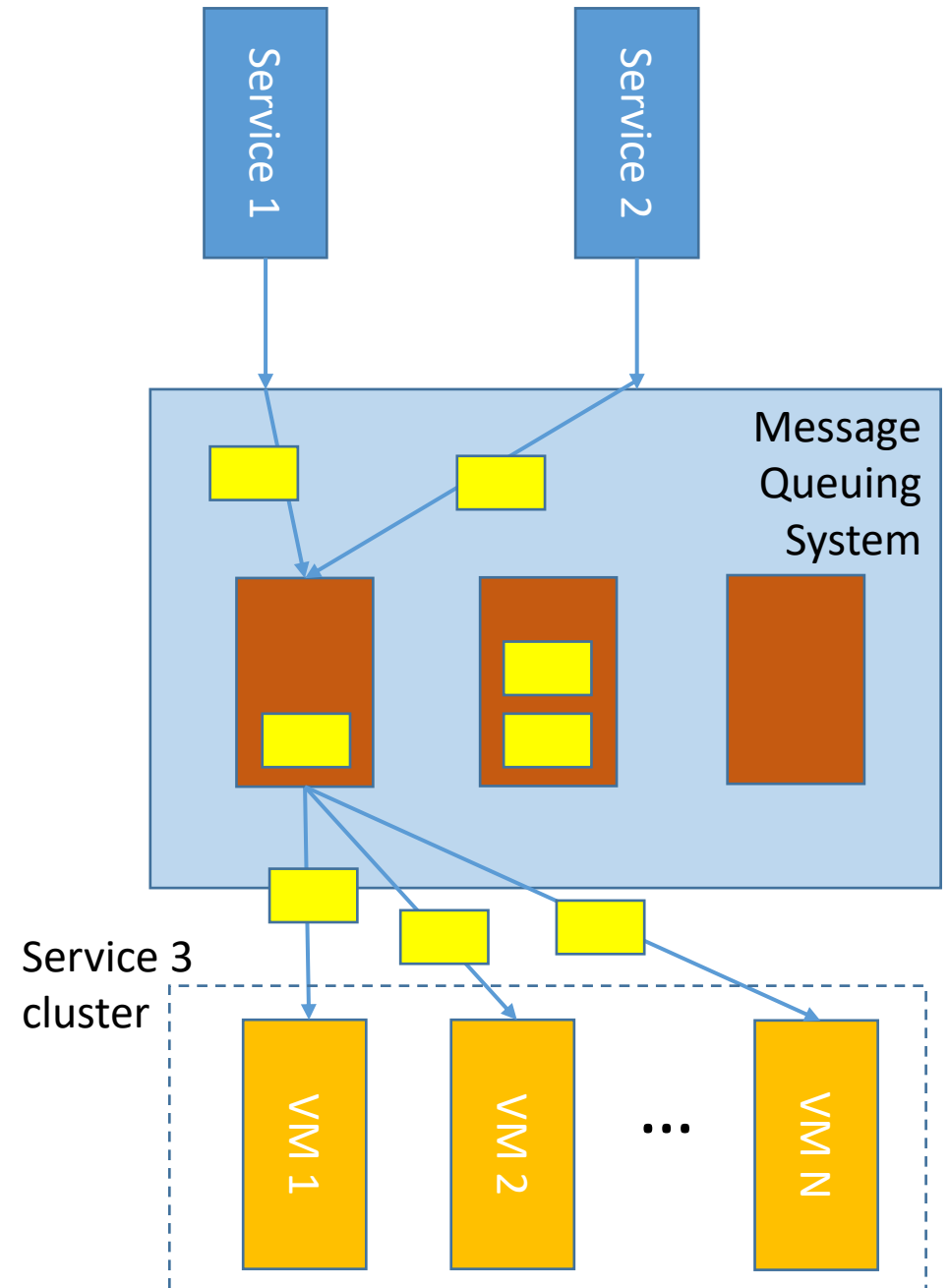
- Each message consists of:
 - a destination (that is, a unique identifier designating the destination queue),
 - metadata associated with the message, including fields such as the priority of the message and the delivery mode
 - body of the message
- The body is normally opaque and untouched by the message queue system. Message selection is normally expressed through predicates defined over the metadata
- The queuing policy is normally first-in-first-out (FIFO), but most message queue implementations also support the concept of priority, with higher-priority messages delivered first
- Consumer processes can also select messages from the queue based on properties of a message
- One crucial property of message queue systems is that messages are **persistent**: *message queues will store the messages indefinitely* (until they are consumed) and will also commit the messages to disk to enable reliable delivery

Message Transformation

- Transformation can be performed on an arriving message
- The most common application of this concept is to transform messages between formats to deal with heterogeneity in underlying data representations
- This could be as simple as transforming from one byte order to another (big-endian to little-endian) or more complex, involving for example a transformation from one external data representation to another.

Scalability

- Message queues are perfect for service communication at the backend of cloud applications, as they can handle the dynamicity that characterize that environment
- For instance, they can handle the varying number of VMs in a tier to scale the system: the VMs that compose the cluster of a tier implementing the same functionalities can receive data from the same queue according to some policy (e.g. load balancing or availability)
- Dynamic situations in which VMs are created dynamically at runtime is handled, as the consumer does not need to exist when the message is generated
- Producers do not have to wait for the Request-Response blocking message exchange

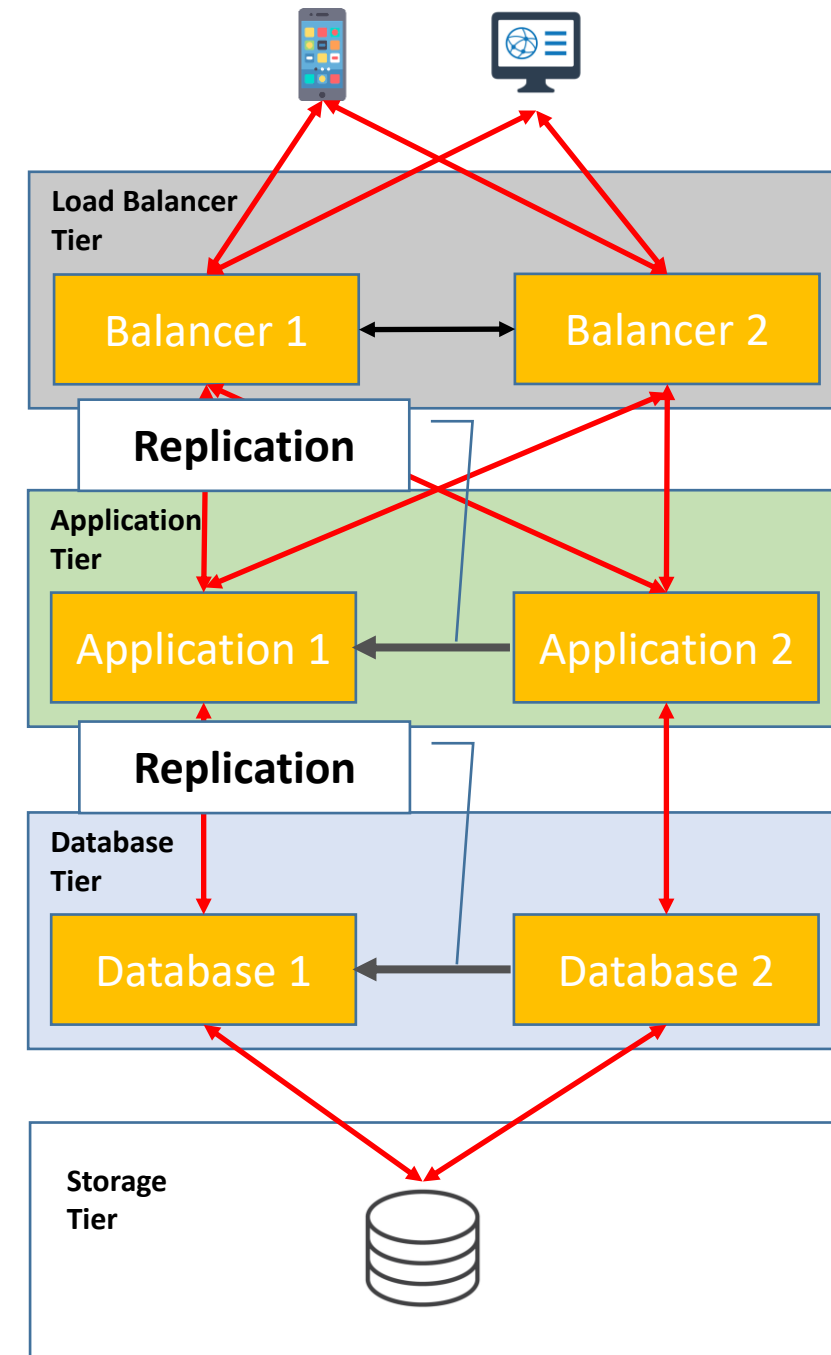


Implementations

- Message Brokers and Message Queuing Systems can be implemented in a centralized and distributed manner
- In a centralized implementation, the broker or the message queuing system is implemented on a single system (e.g. a single VM)
 - This implementation is straightforward, however, it lacks resilience and scalability since it represents a single point of failure and a performance bottleneck
- In a distributed implementation a network of brokers/message queuing systems is adopted
 - It is more complex, as coordination and data replication is required, however it guarantees that resiliency and scalability
- The Advanced Message Queuing Protocol (AMQP) is one of the most widely adopted implementation for Message Queuing Systems

Data Replication

- In cloud applications each tier includes a cluster of VMs implementing the same functionality to ensure **performance enhancements (scalability), availability and fault tolerance**
- Data replication across the VMs of the same tier is required in order to ensure that they all work on the same set of data, so they can provide the same service to users
- Generally, data replication is implemented via message exchange among the VMs of the same cluster
- When large datasets are involved, a shared storage is usually adopted: the shared storage offers a common data set that is shared among all the VMs
- Even in this case, however, the exchange of the data among different VMs is only reduced as exchange of data for coordination is required

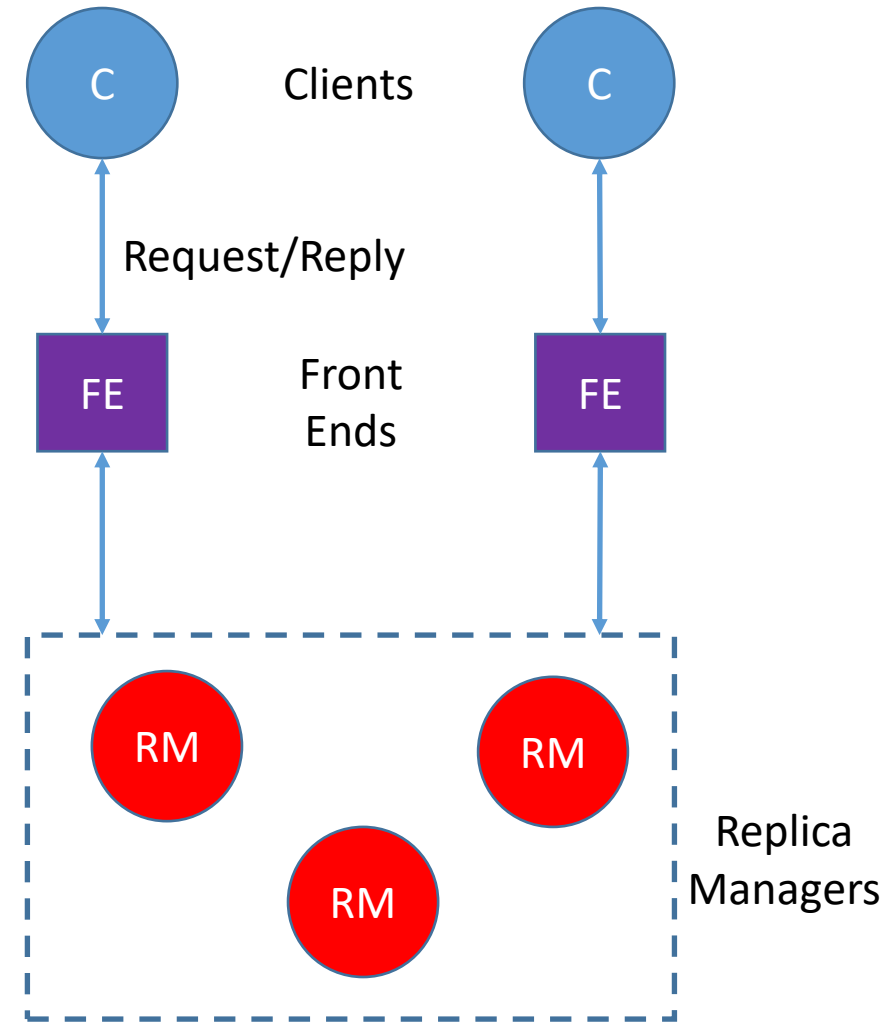


Requirements

- Data replication across different VMs should take into account the following requirements:
 - Replication transparency: data replication should be transparent from users, i.e. they should access different copies of the data without being aware of the replication system
 - Consistency: replicas should be consistent with each other, i.e. an operation performed on any replica should produce the same result
 - Resiliency: the system should be resilient from VM failure (at a certain extent), however, it is assumed that network partitions (outages in the communication) cannot occur. The latter assumption is reasonable whenever the VMs are connected to the same LAN, i.e. they are deployed on the same datacenter. NOTE: *Data replication methodologies that handle network partition will be covered later*

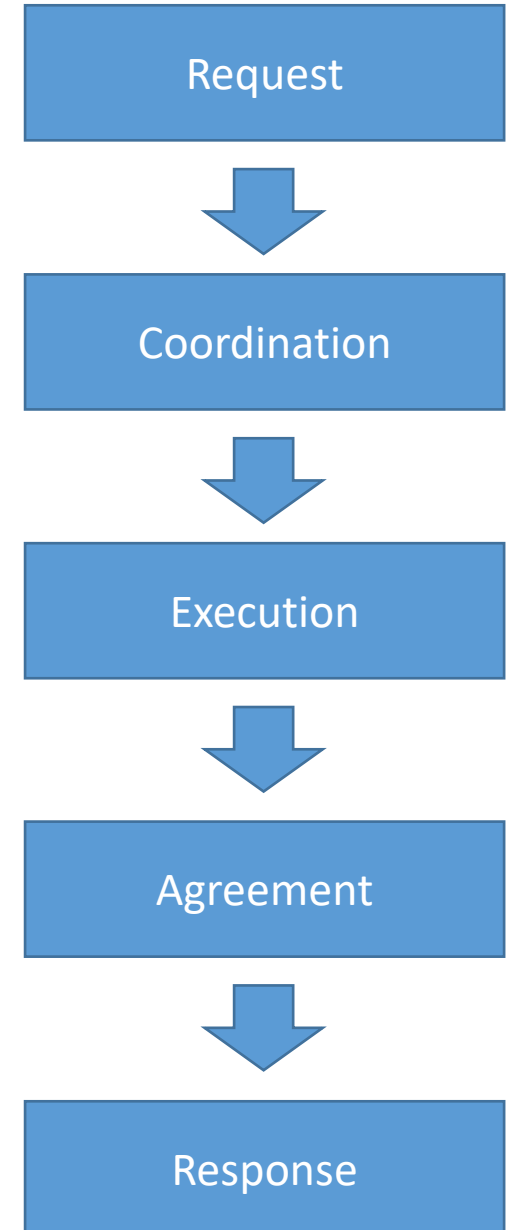
System Model

- The data in the system is a collection of objects, i.e. a file, a class or a set of records. Each logical object is implemented as a collection of physical copies, named replicas
- Each replica is stored in a different replica manager, a different component that stores an instance of the replica (a VM composing the cluster of a tier, in our case)
- In order to ensure consistency, replicas are state machines, meaning that in addition to data, the state machine is also replicated
- Each replica manager accepts update operations on the data in a recoverable manner, so inconsistencies due to failure can be recovered



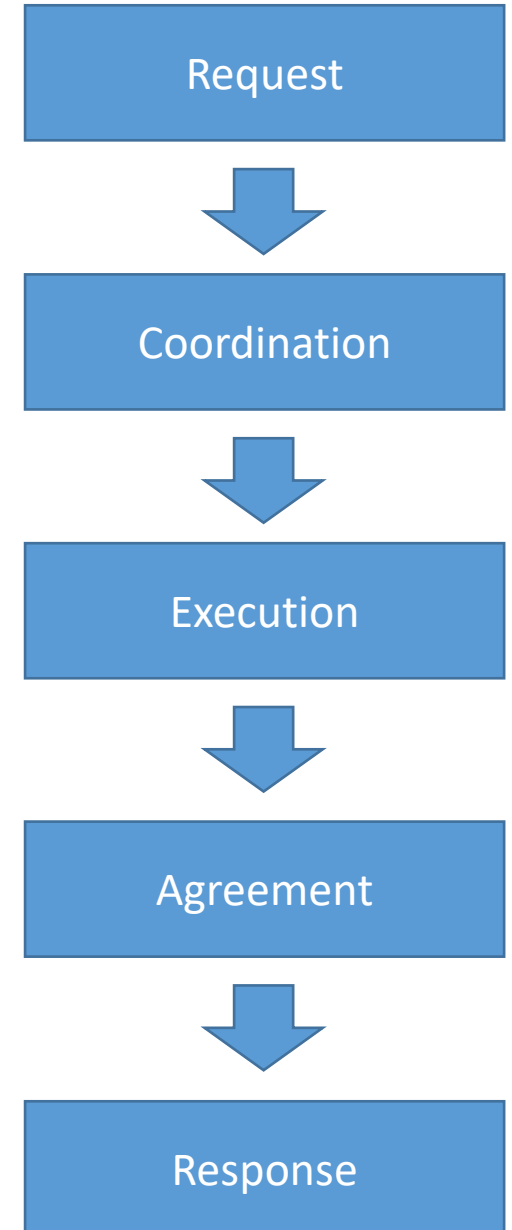
Request handling

- A request can be a **query**, which does not modify the status of the object, or an **update**, which modifies the status of the object
- **In general**, the following phases are implemented by the system to handle a single request from a client:
 1. Request: The request is received by a frontend module that issues the request to one or more replica managers (an instance of the first backend tier)
 - Depending on the implementation, the frontend can communicate with only one single replica manager, which in turn communicate with the other replica managers, or can communicate directly with all the replica managers



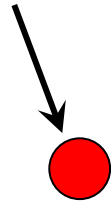
Request handling

2. Coordination: The replica managers coordinate to prepare the execution of the request. They can coordinate to decide if the request can be executed or it must be deferred in order to maintain consistency
3. Execution: The replica managers execute the request
 1. The request is usually executed tentatively, i.e. in a manner that can be undone later if needed, e.g. to recover from faults
4. Agreement: The replica managers reach consensus on the effects of the request (*if needed*). If agreement is reached, the execution of the request is committed; changes (if any) are applied
5. Response: One or more replica manager respond to the frontend
 1. In some systems, only one replica manager sends the response, in others, a collection of replica managers reply

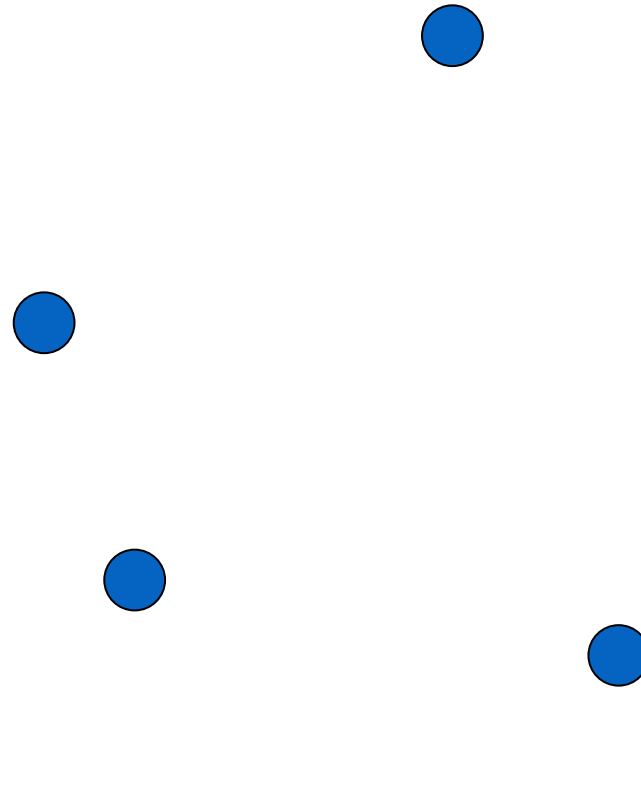


Group/Multicast Communication

Process with a piece of information
to be communicated to everyone




Multiple phases
exploit group
communication among
replica managers (the
group members) for
coordination and data
exchange



Distributed
Group of
Processes
at Internet-
based hosts

Group/Multicast Communication

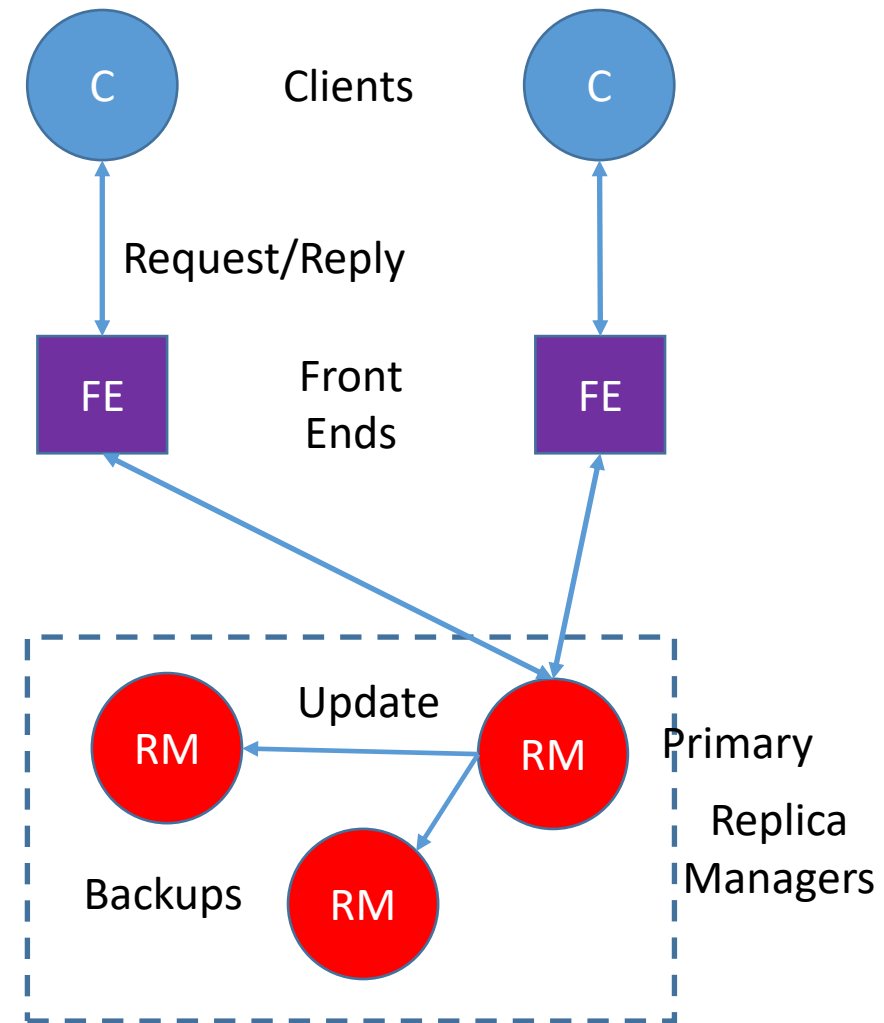
+ Complexity | + Guarantees



- Different Multicast methods are available with different properties, among them:
 - Reliable/Atomic multicast: the message is delivered to every correct member, or to no member at all, losses in the communication are handled, duplicates are removed. No guarantees about order is offered.
 - Total/Casual order multicast: Every member must receive all updates in the same order. If a, b are two updates and a happened before b, then every member must receive a before receiving b. No guarantees on the absolute order (absolute time of reception) across different members is offered.
 - View-synchronous communication: with respect to each message, all members have the same exact view all the time

Primary-Backup Replication

- The passive or primary-backup replication model is adopted to guarantee **fault tolerance**
- At any time, a single replica manager is selected as primary, while the others are backups or slaves
- The frontends communicate only with the primary replica manager, which is responsible for handling the request and sending copies of the updated data to the backups
- If the primary fails, one of the backups are promoted to primary

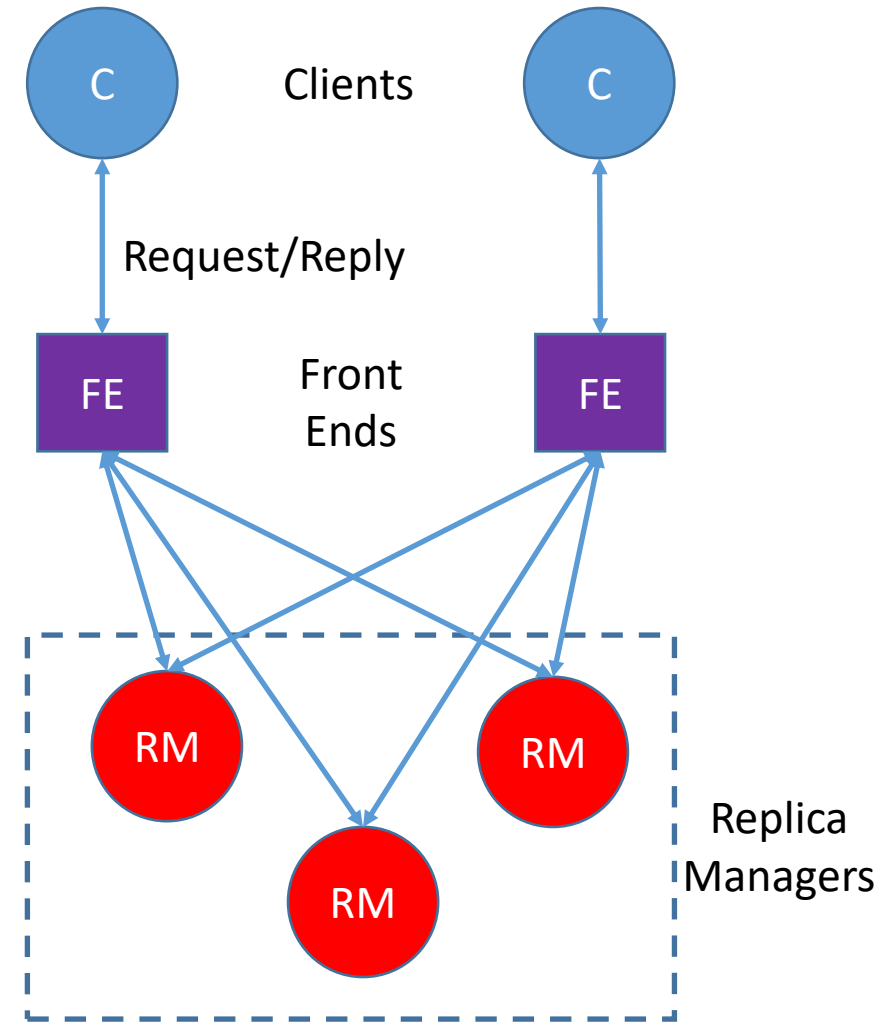


Primary-Backup - Request handling

1. Request: the frontend issues a request, the request contains a unique identifier, which identifies univocally the request
2. Coordination: The primary handles requests atomically, in the order in which they have been received. It checks the unique identifier, if the request has been already executed, the same response is sent back. No coordination with other replica managers is needed
3. Execution: The primary executes the request and stores the response
4. Agreement: If the request is an update, the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgment. For this kind of communication atomic multicast is sufficient
5. Response: the primary responds to the frontend, which hands back the response to the client

Active Replication

- Another different approach to guarantee high availability is active replication
- Replica managers are organized in groups, frontends multicast their requests directly to the group so all the replica managers in the group can process the request independently and reply (identically)
- If a crash occurs in any of the replica managers, there is no impact on the performance as the remaining replica managers can reply



Active Replication - Request handling

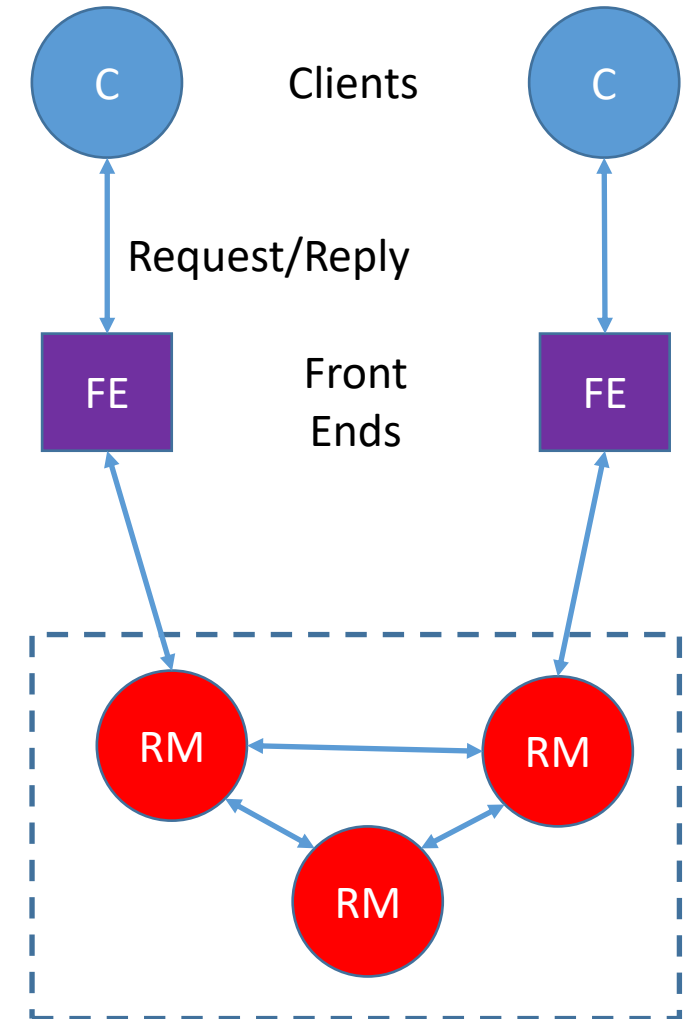
1. Request: The frontend attaches a unique identifier to the request and multicast it to all the replica managers.
2. Coordination: No coordination is needed in this phase, the multicast method takes care of sending the request to all the replica managers. The method adopted has to be a **total order multicast** to ensure that requests are received in the same order by all the replica managers
3. Execution: Every replica manager executes the request. Since the requests are delivered in total order, replica managers process the request identically
4. Agreement: Thanks to the total order multicast, there is no need for agreement
5. Response: Each replica manager sends its response to the frontend. Considering that the multicast algorithm is a total ordered algorithm the frontend can forward the first response received

A different approach

- Passive/active approaches focus on providing high availability:
 - In the passive approach, backup replicas do not take part at handling users request, consequently the load is all on the primary replica, while in the active approach the replicas serve the replicas in a redundant manner
- A different approach has been defined to provide high availability and scalability at the same time:
 - Different replica managers are involved in handling requests from the frontend (not just one or all handling the same request), in parallel, thus maximizing utilization
 - Replica managers do not propagate updates in a prompt manner (as soon as they are available and before passing back control to the client), updates are deferred when possible, while the control is given back to the client as soon as possible (even before propagating the update to other replica managers)

The Gossip architecture

- In the gossip architecture, the replica managers exchange gossip messages periodically in order to convey the updates they have received from clients
- Clients are assigned to specific FE instances for load balancing, even though they can change, if the assigned instance crashes
- The service provides two basic types of operations:
 - **Query**, which are read-only operations
 - **Update**, which modifies the state of an object. The update operation does not read the state
- The gossip system makes two guarantees:
 - *Each client obtain a consistent service over time*, i.e. in answer to a query, a replica manager provides a response with data that reflects the last updates that the client has observed so far (its own updates)
 - *The consistency between replicas can be relaxed in time* due to the frequency of the exchange of gossip messages, i.e. a client receive an old (but consistent) information when a query is issued. **Sequential consistency is guaranteed**

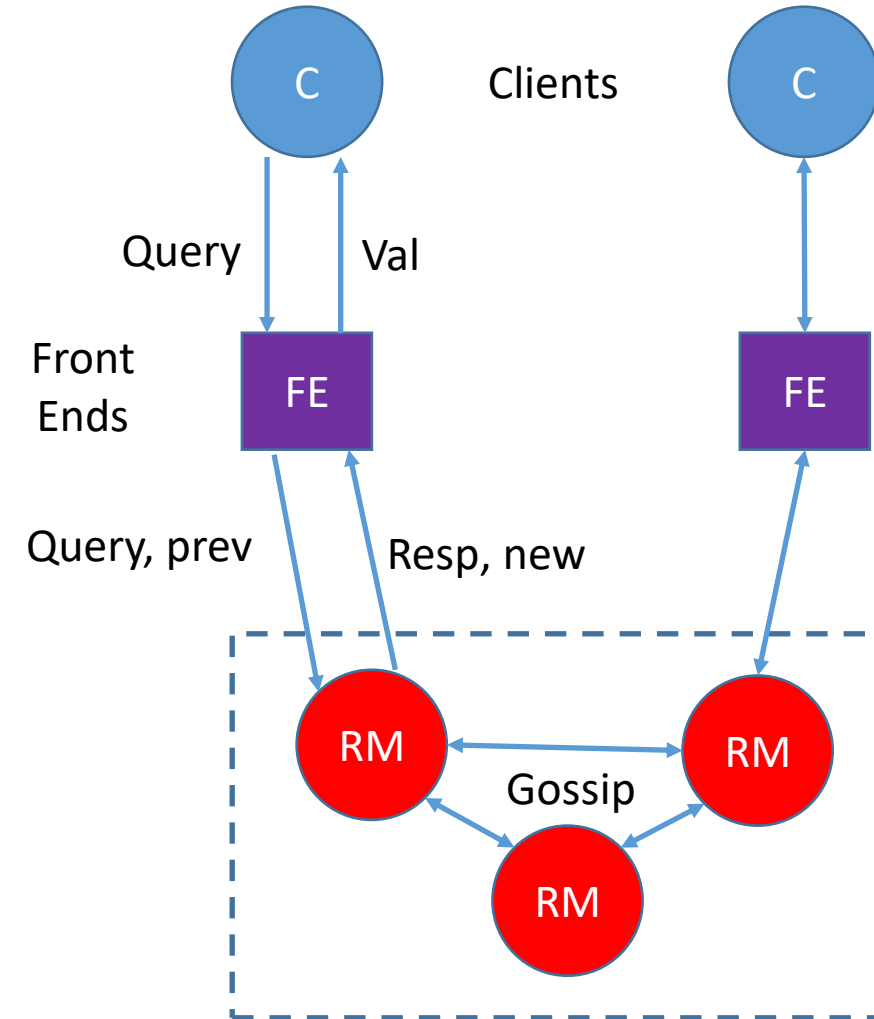


Gossip Architecture - Request handling

1. Request: The front end normally sends requests to only a single replica manager at a time
2. Coordination: The replica manager that receives the request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages. No other coordination between replica managers is involved
3. Execution: The replica manager executes the request
4. Agreement: The replica managers update the data by exchanging gossip messages, which contain the most recent updates they have received. The updates are exchanged in a lazy fashion, i.e. gossip messages are exchanged only occasionally, after several updates have been collected, or after a certain amount of time
5. Response: If the request is a query, the replica manager replies after the execution, if it is an update, it replies right after receiving it

Timestamps

- In order to keep ordering of requests, timestamps are adopted (to this aim FEs must have their clock synchronized)
- Specifically, each FE keeps a vector of timestamps that reflect the version of the latest data values accessed
- This timestamp is sent to the replica manager (the field prev) together with the query
- The replica manager supplies a new timestamp with the response (the field new)
- Each returned timestamp is merged with the previous timestamp value



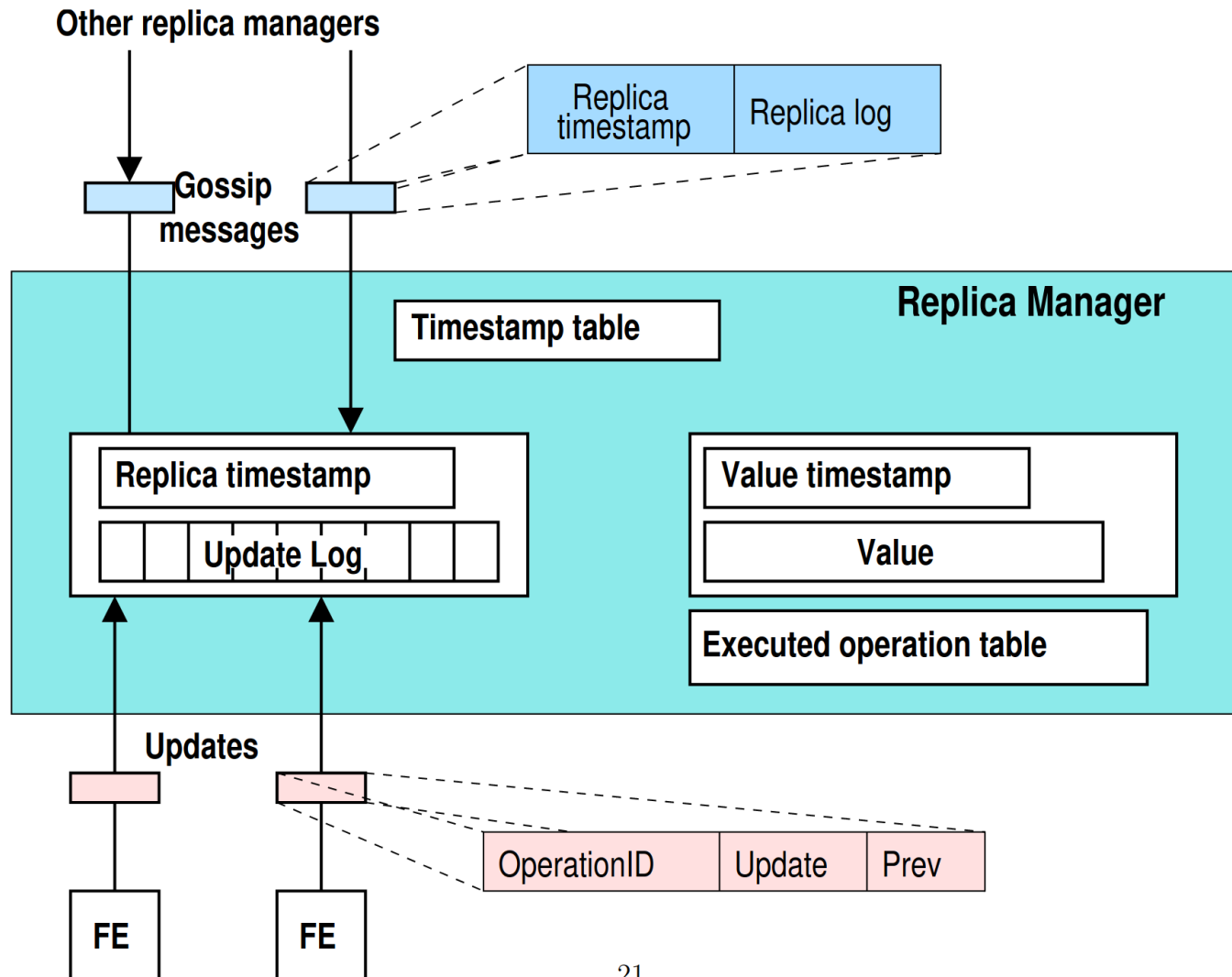
Processing Query

- The timestamp associated with the query request reflects the latest version of the value that the frontend has read or submitted as update
- For this reason, every time a query is received, the replica manager has to return a value that is at least as recent as the value provided with the query
- For this reason, if the timestamp of the latest local replica's update is oldest than the timestamp provided ($valueTS < prev$) the request is deferred and kept on a list of pending operation, in order to wait for a missing update
- Once the query can be executed, the replica manager returns the *valueTS* of the last value update to the frontend

Processing Updates

- When a replica manager receives an update request from a FE it first analyzes the request ID to check if the request has been received and processed already (the frontend uses the same ID each time)
- If the request is not duplicated, the replica manager creates a unique timestamp by incrementing the prev value received from the FE and then it sends back the response with the updated timestamp
- It also adds a new entry to the local log record of pending update operations
- If the same timestamp condition of the query is satisfied ($prev \leq valueTS$) the update operation is committed, otherwise it is deferred until it is satisfied (every time a next gossip update is received the condition is verified)

Replica Manager



Example - ZooKeeper

- ZooKeeper is a high-available service usually exploited for (small) data exchange among different services
- It can be used to store configuration information, create a distributed directory or a naming system
- ZooKeeper is a replicated service, it requires to operate that a majority of nodes are not crashed to progress
- Crashed servers are excluded automatically, if they recover can rejoin
- It uses a primary-backup scheme to operate and implements an atomic broadcast algorithm named Zab to operate
- WebSite: <https://zookeeper.apache.org>