

263-3855-00: Cloud Computing Architecture

Section 08: Communication APIs

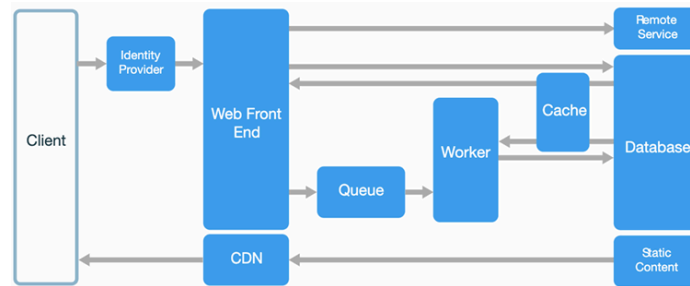
Swiss Federal Institute of Technology Zurich (ETH Zürich)

Last Edit Date: 05/31/2025

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

How distributed components are connected plays a key role in performance, architecture, reliability, availability, etc.



I. Remote procedure call (RPC)

1. Origins of RPC

- System went from 1-tier to 2-tier architecture and beyond, due to
 - Increasing availability of computers
 - Increasing availability of high-performance networks
- Communicating at the level of sockets is complicated and not efficient from a programming perspective
- How to enable communication while matching the high level of abstraction available in programming languages?
 - Make remote procedure call look as much as possible like local procedure call
- In 1984, Andrew D. Birrell and Bruce Jay Nelson proposed a way of implementing procedure calls, published in ACM transactions on computer systems, which is still the system architecture we use today.

II. The basics of RPC

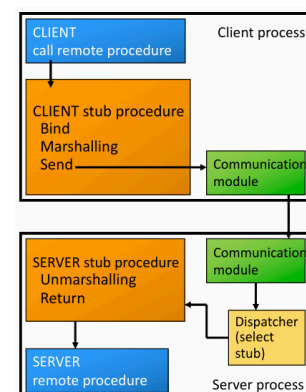
RPC turns a procedure call into a call to a remote procedure:

- The programmer still thinks in terms of calling a procedure (function / method)
- Data passing is hidden behind the call arguments
- A distributed program looks like a normal program
- All the complexity hidden in the “stub”

Key challenges for implementing RPC

Need to decide

- Integration of remote calls into programs, with client and server possibly in different programming languages
- How a caller determines the location and identity of the callee (i.e., binding)



- Semantics of address-containing arguments (when no shared address space)
- Semantics of a call in case of machine or communication failures

RPC key steps

- Call a procedure
 - The call is replaced by a call to a “stub” that presents the same interface as the procedure but manages the remote call
 - The stub and the interfaces to link to must be generated beforehand
- Bind
 - Find the remote server implementing the procedure: IP address, port, etc.
 - This requires having a Name and Directory Service
- Serialization / Marshalling
 - Map the call parameters to a format suitable for transmission (string of bits)
 - This implies some sort of standard intermediate representation

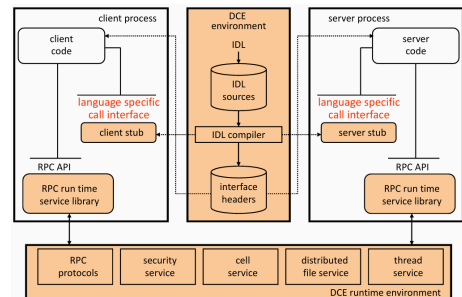
Why do we need data serialization for RPCs?

- An object in memory has a machine-specific binary representation
 - Threads within a single process have the same view of what is in memory
 - Easy to compute offsets into fields, follow pointers, etc.
- In the absence of shared memory, externalizing an object requires us to turn it into a sequential sequence of bytes
 - Serialization / Marshalling: Express an object as a sequence of bytes
 - Deserialization / Unmarshalling: Reconstructing the original object from its marshalled form at destination

From the RPC concept to an RPC system

RPC runtime environment needs to implement several services:

- Name and directory service: servers register their procedures, clients look for the addresses of procedure that can be invoked by others and then post the necessary information in the directory
- Interface definition language (IDL): how to declare a procedure as a remote procedure that can be invoked by others and then post the necessary information in the directory
- Serialization: an expensive and important step done by the stub (since data representation in one machine not necessarily same on a different machine)
- Plus security, system management, etc.



RPC in perspective

Advantages	Disadvantages
• RPC provided a mechanism to implement distributed applications in a simple and efficient manner • RPC followed the programming techniques of the time (procedural languages) and fit well with popular programming languages (C), which facilitates its adoption • RPC allowed the modular and hierarchical design of large distributed systems: (1) client and server are separate; (2) the server encapsulates and hides the details of the backend system (e.g., DBs)	• RPC is not a standard, it is an idea that has been implemented in different ways (not necessarily compatible) • RPC is a low-level construct to build distributed systems but does not solve many of the problems that distribution creates • RPC was designed for client / server architecture, which reflected the hardware at the time (small terminals connected to a mainframe). As hardware and networks evolved, more flexibility was needed

CORBA

- CORBA was an attempt in 1990s to standardize RPC for object oriented languages
- Not only RPC mechanism but many services attached to the standard
- Designed for local area networks in 1990s (traffic directly over TCP), which was not suitable for WWW which builds on HTTP

RPC system issues

- Should a remote procedure be transparent (identical to a local procedure)?
 - Yes: more intuitive for ease of use?
 - **No**: increase programmer awareness - remote procedure call is always slower than a local call. Also might fail if remote server not available
- Should the server location be transparent?
 - Yes: good for flexibility and fault tolerance
 - **No**: increase programmer awareness, easier design, less overhead

Need programmers to be aware of remote calls

- Making remote calls look to programmers exactly like local procedure calls is problematic
 - Especially given the scale of today's systems and the cloud
 - Many heated discussions at the beginning of RPC (programming elegance and simplicity vs. systems reality). Today, it is known to be a bad idea.

- Key insight
 - It is not the same to make a local call (guaranteed to return, very low latency) as a remote one (not guaranteed to return, high latency, potential long timeouts). The logic needed around the call is completely different

Synchronous calls

RPC is a synchronous call: both caller and callee must be available when the call is made

- Can be very efficient
- Result in tight coupling: lower availability (if one of them fails, the whole system fails)
- Results in high latency: caller is blocked until call returns (this led to the introduction of multi-threading in systems)
- Difficult to recover from failures as semantics are not well defined:
 - At least once: the call will happen but might happen several times
 - At most once: the call might happen or not but if it does, it happens only once
 - Exactly once: the call will happen and will happen only once

Augmenting RPC with failure semantics

- Attach a unique identifier to each call and “remember” calls by persisting identifiers
- For at least once semantics: the client remembers the call it made and if the call does not come back (or client failed), makes the call again until it succeeds
- For at most once semantics: the server remembers the call made, so even if server fails, it ensures repeated calls are not executed twice
- For exactly once semantics: both server and client remember the call

Serialization is needed but expensive

- Serialization is the result of coupling the interface for remote communication to a programming language
 - Call parameters are actually differently on different machines and across languages
 - They can be represented differently on different machines and across languages
 - They need to be turned into a string of bytes to be sent over the network
 - Needs to be done on sending and receiving side
- Serialization accounts for 6% of the total CPU cycles in Google data centers

What has the cloud changed?

- Compared with early days of RPC, the cloud is a very different environment
 - Multi-language (cannot assume everybody is using the same language)
 - Scale (the size of the systems is much larger and so are the consequences of tight coupling and synchronicity)

- Communication protocols (today HTTP instead of TCP/IP as used in RPC)
- A better understanding of architecture (REST interface)
- RPCs are still heavily used today, but implementations have evolved
 - For example, gRPC runs on HTTP/2, supports data streaming, uses the protobuf binary serialization format (as opposed to JSON/XML), becoming more efficient and scalable

II. REST interfaces and HTTP

While highly efficient, RPC becomes problematic at large scales and in the presence of failures

- Less of an issue at the time that RPC was developed
- A big problem at internet / cloud scales, with resources under different scopes of control

RPC ran into trouble with the WWW, as one can call server from anywhere

- RPC was tied to TCP/IP
- RPC was tied to programming languages and programming language paradigm
- Implementations focused mostly on local area networks (e.g., not dealing with firewalls)

HTTP provides a basis for a different way of implementing distributed calls

1. What is REST?

- REST is a general concept. Its most common implementation is HTTP.
- REST APIs describe resources (instead of procedures), which can be anything: documents, data, objects, services, etc.
- Resources have a unique identifier (URI, similar to a URL)
- Data is exchanged using character formats (representations) using standards like JSON
- Simple interfaces: GET, PUT, DELETE, ...
- Interaction is stateless (no dependencies between calls, no assumptions of ordering of calls)

2. Why REST?

Access more than programs

- RPC was designed to call a program from another program, which made sense when the only thing communicating were programs
- Today, we might want to access many more things than a program without having to think about them as being a program:
 - Access a document as a web page
 - Insert data into a system
 - Call a service
- REST APIs describe resources (instead of procedures), which can be anything: documents, data, objects, services, etc.

- Making everything look like an URL (URI), which unifies access to all these objects

Decouple from implementations

- Resources have a unique identifier (URI, similar to an URL)
- Using an URI removes dependencies from concrete implementations of RPC and allows to make calls across completely different systems and domains (in essence, the interface is an internet resource)
- The role of the binder and the Name and Directory Service is replaced by existing infrastructures (DNS)
- It decouples the service from the implementation and the transport protocol (unlike most RPC implementations)

Standardized representations

- Data is exchanged through character formats (representations) using standards like JSON
- Instead of using proprietary exchange formats, serialization / marshalling is done using a standard representation such as XML or JSON that are well documented and understood
- This enables the development of standard tools for processing data and also makes it easier to keep the data in that format rather than translating it back and forth all the time

Simplify the interfaces

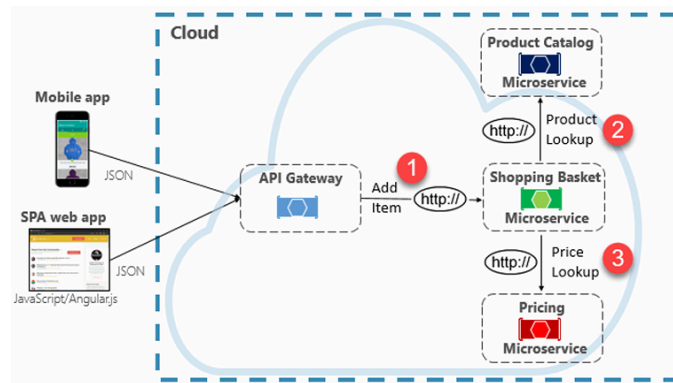
- Simple interfaces: GET, PUT, POST, DELETE, ...
- In RPC, the interface is procedures. As services evolves, the number of procedures becomes very large and, thus, difficult to maintain and use
- Based on the notion of resources, the HTTP interface is very simple (GET, POST, PUT, DELETE) regardless of what operation is being performed
- The actual operation is described in the resource and not exposed as a procedure. For example, instead of calling a procedure to create an account, one POSTs a document containing the information for the new account

Scalability

- Interaction is stateless (no dependencies between calls, no assumptions of ordering of calls)
- RPC lends itself to developing applications logic that is intertwined between the client and the server (after all, we are just using procedures)
- REST forces the exchange to become explicit and, if implemented correctly, allows each call to be treated independently
 - Rather than keeping state on the server / client, pass the state as part of the resource being exchanged

- We can then use different and parallel servers to deal with requests even if they come from a single client (this might be good or bad)

Today, cloud systems tend to use REST calls (HTTP) between services rather than using RPC. This makes it easier to scale, route requests, implement reliable systems, etc. These calls are still synchronous.



3. gRPC - RPC meets REST

- Over time, implementations of RPC have evolved to adopt ideas from REST such as gRPC (open source, initially from Google)
- gRPC uses a standardized representation for serialization (protocol buffers). Several such representations have been created by cloud providers:
 - Ion (Amazon)
 - Apache Thrift (Meta)
 - Microsoft Bond
- Operates on top of HTTP/2 instead of TCP/IP
- Often used to implement performance critical systems in the cloud (real time, low latency, high throughput, ...)

III. Design patterns to manage communication

1. REST is not enough

- RPC became problematic when there was a proliferation of services, languages, and networks (RPC was developed for one program to call another program on a remote node)
- REST becomes problematic when used in large-scale multi-tier settings as interaction becomes cumbersome and requires too many steps in environments where network latency matters (REST was based on the web and HTTP, not data centers)
- Design patterns have been developed to cope with these challenges

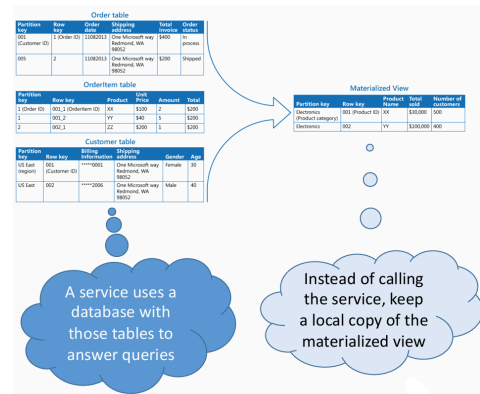
Long chains of calls

- Systems today are very complex, involving many services and microservices

- REST calls are still synchronous and add latency
- Long chains of calls make the system unreliable, slow, and difficult to manage

2. Design pattern 1: Materialized view

- Most services just retrieve data and provide information to the caller
- Instead of relying on a service, the caller can cache a copy of the data it needs and use it instead of calling a service
- Often in the form of a materialized view that presents the information exactly like the service needs it

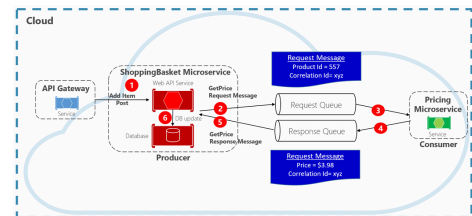


3. Design pattern 2: Service Aggregator

- Part of the problem is that independent services can create a complex web of calls across services
- Instead, implementing a service that enforces the sequence of calls by making the calls from a centralized place (the aggregator)
- Calls are still HTTP but there are no dependencies between the services

4. Design pattern 3: Message queues

- Reliability and performance issues can be ameliorated by decoupling the services through queues:
 - Messages instead of calls
 - Some calls do not need a response (commands)
- Adds latency but makes the services completely decoupled



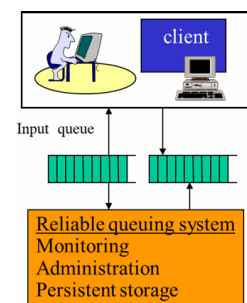
IV. Messaging and queuing systems

- Message queuing systems are nothing new and predate the cloud:
 - Many commercial and open source systems: IBM's MQ-Series, Java Message Service (JMS), ...
 - Now many systems in the cloud: Amazon MQ, Azure Queue Storage, Amazon SQS, Apache Kafka, ...
- The basics mechanism are similar as well as the principles behind them

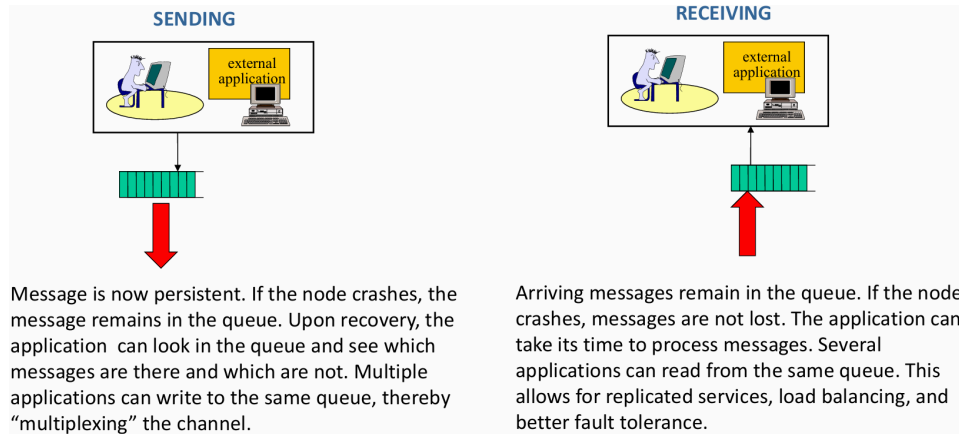
1. Basic functionality

A message queuing system typically

- Provides queue for clients to PUT and GET messages
- Stores the messages persistently to guarantee they survive failures



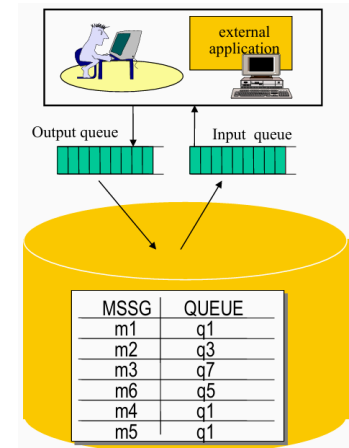
- Makes the insertion and deletion of messages transactional
- Monitor queues and accesses for access control, scalability, routing, etc.
- Manages the creation, deletion, and access to messages and queues



2. Implementations

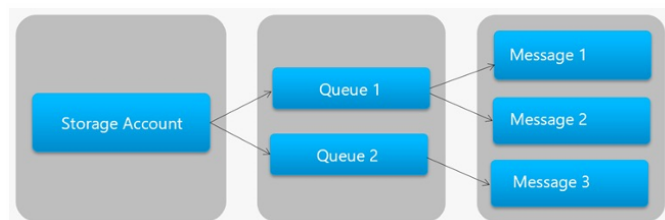
Simple implementation

- Persistent queues can be implemented as part of a database since the functionality needed is exactly that of a database
 - A transactional interface
 - Persistence of committed transactions
 - Advanced indexing and search capabilities
- Thus, messages in a queue become simple entries in a table. These entries can be manipulated like any other data in database so that applications using the queue can assign priorities, look for messages with given characteristics, trigger certain actions when messages of a particular kind arrive



Azure Storage Queues

- A simple queuing service implemented on top of cloud storage
 - Pay only for the storage
 - Reliable and persistent
 - Large capacity
 - Access through HTTP
 - No message ordering
 - Persistence limited to 7 days
 - No support for duplicate detection nor transactional access

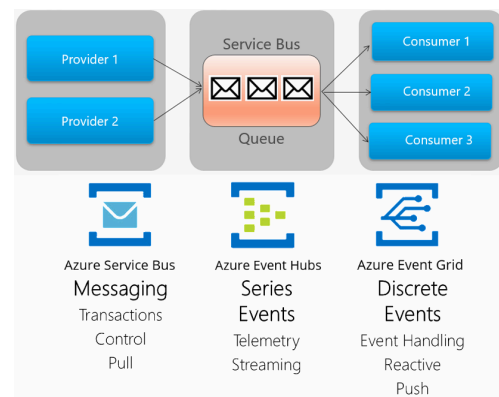


Amazon SQS

- Simple queue system
 - Two types of queues:
 - Best effort ordering and at-least-once delivery
 - FIFO queues with exactly once semantics
 - No limitations on volume
 - Multiple copies of the messages made across availability zones (automatic)
 - Elastic deployment (queues are dynamically created and scaled automatically)
 - Charged per 1 million requests (standard queue cheaper than FIFO queues) and GB of data read out

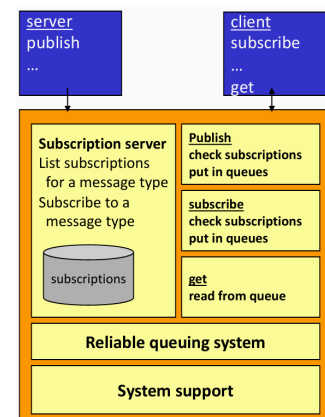
Azure Service Bus Queues

- Based on a message broker, it provides the typical features of a message queuing system
 - FIFO delivery
 - Transactional access
 - Duplicate detection
 - Supports queue partitioning across several message brokers (for scalability)
 - Session IDs used to correlate messages that belong together (correlation IDs)
 - Size of the queue is limited
 - Charged on a base cost and per operation

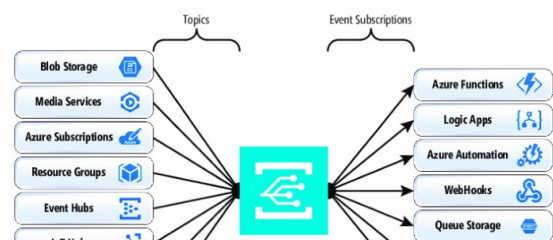


Publish subscribe: event-based systems

- Standard client / server architectures and queuing systems assume point-to-point interaction
- In many situations, it is more useful to implement systems where the interaction is based on announcing given events:
 - Service publishes messages / events of some type
 - Clients subscribe to different types of messages / events
 - When a service publishes an event, the system looks at a table of subscriptions and forwards the event to interested clients by transferring the event's message into a queue for that client



Azure Event grid



- These event systems implement a push model where the message (event) is pushed to the subscriber
- Allows to implement notifications that are sent to many destinations
- Guarantees delivery within a 24 hour period, automatic retry
- No transaction, no ordering, no sessions

Takeaways

- Building distributed applications requires a communication infrastructure
- Cloud provides conceptually similar communication as distributed systems before the cloud: RPC, messages, events
 - But at a large scale
 - Adapted to systems and services provided in the cloud
 - With a wider range of use cases
- Use RPC for performance low latency
- Use REST for scalability, more standardized protocols
- Use Messaging for reliability, decoupling of components
- Use Events for massive scale, push notifications

V. Related readings

1. (Optional) [A Cloud-Scale Characterization of Remote Procedure Calls](#)