Communication in Enterprise Architecture

Gustavo Alonso
Systems Group
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
alonso@inf.ethz.ch
We have already discussed how to organize the different layers of a distributed system.

This week we will discuss how to connect the different layers and modules once we have decided to distribute them.

The same principle applies:
- Functionality is added through layers of indirection.
- The more indirection, the more functionality can be achieved.
- To get performance, indirection has to be bypassed when the loss of functionality can be tolerated.

How the different parts of a system interact affect the way the system is developed and what it can do in practice.

There is no problem in system design that cannot be solved by adding a level of indirection.

There is no performance problem that cannot be solved by removing a level of indirection.
RPC Call semantics

RPC is one to one
Possible semantics are:
- **Maybe**: no guarantees.
- **At least-once**: the procedure will be executed if the server does not fail, but it is possible that it is executed more than once.
- **At most-once**: the procedure will be executed either once or not at all.

Enforcement of these semantics involves additional mechanisms (message counters, message ids, remembering message ids, etc.)

More complex semantics require transactional RPC to be able to extend the properties to sets of calls

RPC results in tightly coupled systems
- Both ends must work
- Knowledge of both ends is needed to program an application

With RPC is difficult to implement any other form of communication than one-to-one:
- N to N
- Many to one
- One to many
- Content driven distribution
- Filters based on content
- Call routing

RPC hides the communication channel behind a programming language construct. That is its main advantage and its key disadvantage.
Blocking or synchronous interaction

- Traditionally, information systems use blocking calls (the client sends a request to a service and waits for a response of the service to come back before continuing doing its work)

- Synchronous interaction requires both parties to be “on-line”: the caller makes a request, the receiver gets the request, processes the request, sends a response, the caller receives the response.

- The caller must wait until the response comes back. The receiver does not need to exist at the time of the call (TP-Monitors, CORBA or DCOM create an instance of the service/server/object when called if it does not exist already) but the interaction requires both client and server to be “alive” at the same time.

- Because it synchronizes client and server, this mode of operation has several disadvantages:
  - connection overhead
  - higher probability of failures
  - difficult to identify and react to failures
  - it is a one-to-one system; it is not really practical for nested calls and complex interactions (the problems becomes even more acute)
Overhead of synchronism

- Synchronous invocations require to maintain a session between the caller and the receiver.
- Maintaining sessions is expensive and consumes CPU resources. There is also a limit on how many sessions can be active at the same time (thus limiting the number of concurrent clients connected to a server).
- For this reason, client/server systems often resort to connection pooling to optimize resource utilization:
  - have a pool of open connections
  - associate a thread with each connection
  - allocate connections as needed
- Synchronous interaction requires a context for each call and a context management system for all incoming calls. The context needs to be passed around with each call as it identifies the session, the client, and the nature of the interaction.

Context is lost
Request should be retried!!
Failures in synchronous calls

- If the client or the server fail, the context is lost and resynchronization might be difficult.
  - If the failure occurred before 1, nothing has happened
  - If the failure occurs after 1 but before 2 (receiver crashes), then the request is lost
  - If the failure happens after 2 but before 3, side effects may cause inconsistencies
  - If the failure occurs after 3 but before 4, the response is lost but the action has been performed (do it again?)

- Who is responsible for finding out what happened?

- Finding out when the failure took place may not be easy. Worse still, if there is a chain of invocations (e.g., a client calls a server that calls another server) the failure can occur anywhere along the chain.
Two solutions

**ENHANCED SUPPORT**

- **Client/Server systems and middleware platforms** provide a number of mechanisms to deal with the problems created by synchronous interaction:
  - **Transactional interaction**: to enforce exactly once execution semantics and enable more complex interactions with some execution guarantees
  - **Service replication and load balancing**: to prevent the service from becoming unavailable when there is a failure (however, the recovery at the client side is still a problem of the client)

**ASYNCHRONOUS INTERACTION**

- Using asynchronous interaction, the caller sends a message that gets stored somewhere until the receiver reads it and sends a response. The response is sent in a similar manner.
- Asynchronous interaction can take place in two forms:
  - **non-blocking invocation**: (a service invocation but the call returns immediately without waiting for a response, similar to batch jobs)
  - **persistent queues**: (the call and the response are actually persistently stored until they are accessed by the client and the server)
Message queues
Message queuing

- Reliable queuing turned out to be a very good idea and an excellent complement to synchronous interactions:
  - Suitable to modular design: the code for making a request can be in a different module (even a different machine!) than the code for dealing with the response
  - It is easier to design sophisticated distribution modes (multicast, transfers, replication, coalescing messages) and it also helps to handle communication sessions in a more abstract way
  - Message-based queuing systems offer a more natural way to implement complex interactions between heterogeneous systems
Queuing systems

- Client
- Input queue
- Output queue
- Reliable queuing system
  - Monitoring
  - Administration
  - Persistent storage
- External application
- Input queue
- Output queue
Transactional queues

- Persistent queues are closely tied to transactional interaction:
  - to send a message, it is written in the queue using 2PC
  - messages between queues are exchanged using 2PC
  - reading a message from a queue, processing it and writing the reply to another queue is all done under 2PC

- This introduces a significant overhead but it also provides considerable advantages. The overhead is not that important with local transactions (writing or reading to a local queue).

- Using transactional queues, the processing of messages and sending and receiving can be tied together into one single transactions so that atomicity is guaranteed. This solves a lot of problems!
Problems solved (I)

Message is now persistent. If the node crashes, the message remains in the queue. Upon recovery, the application can look in the queue and see which messages are there and which are not. Multiple applications can write to the same queue, thereby “multiplexing” the channel.

Arriving messages remain in the queue. If the node crashes, messages are not lost. The application can now take its time to process messages. It is also possible for several applications to read from the same queue. This allows to implement replicated services, do load balancing, and increase fault tolerance.
Problems solved (II)

- An application can bundle within a single transaction reading a message from a queue, interacting with other systems, and writing the response to a queue.
- If a failure occurs, in all scenarios consistency is ensured:
  - if the transaction was not completed, any interaction with other applications is undone and the reading operation from the input queue is not committed: the message remains in the input queue. Upon recovery, the message can be processed again, thereby achieving exactly once semantics.
  - If the transaction was completed, the write to the output queue is committed, i.e., the response remains in the queue and can be sent upon recovery.
  - If replicated services are used, if one fails and the message remains in the input queue, it is safe for other services to take over this message.
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 a 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 …
Queues in practice

- To access a queue, a client or a server uses the queuing services, e.g.,:
  - put (enqueue) to place a message in a given queue
  - get (dequeue) to read a message from a queue
  - mput to put a message in multiple queues
  - transfer a message from a queue to another

- In TP-Monitors, these services are implemented as RPC calls to an internal resource manager (the reliable queuing service)

- These calls can be made part of transaction using the same mechanisms of TRPC (the queuing system uses an XA interface and works like any other resource manager)
Many systems out there

- Apache ActiveMQ
- Jboss Messaging
- WebSphere MQ (IBM)
- Oracle Advanced Queuing
- Microsoft Message Queuing
- JORAM
- Java Message Service (API for messaging)
- ...
Other forms of interaction
RPC as basic building block

- PUT
- GET
- Publish subscribe system

©Gustavo Alonso, ETH Zürich.
Interactions

RPC/RMI supports request-replay interaction
- blocking
- interface based
- 1-to-1 interaction

Requires a similar programming paradigm on both ends

Enforces a tightly couple architecture although additional middleware layers help to reduce dependencies
- Request is always to a server
- Reply is always to sender

Almost all forms of interaction are implemented on top of RPC

Queues add an additional layer of indirection that allows for more sophisticated routing
- 1-to-1
- 1-to many
- many-to-1
- many-to-many

They decouple the interaction from the modules communicating:
- No need to know the recipient (there might not be any)
- They turn the interaction into something tangible that can be manipulated
Advanced functionality

- Queues allow to implement complex interaction patterns between modules:
  - 1-to-1 interaction with failure resilience
  - 1-to-many (multicast: put in a queue and then send from this queue to many other queues) this is very helpful for “subscriptions”. The fact that the queues are implemented in the database even helps with performance since the logic for distribution can be embedded in the database itself
  - many-to-1 many modules send their request to a single module that can then assign priorities, reorder, compare, etc.
  - many-to-many as in replicated services for large amount of clients

- In some cases queues are being used for interactions that are also on-line. If the queues are fast enough (like in a cluster) one can take advantage of the properties of queues at the expense of performance. Building computer farms becomes easier since messages are one more element that can be moved, copied and stored.

- Incorporating queues into databases provides databases with a very powerful tool for designing distributed applications (TP-light).
Message brokers

- Message brokers add logic to the queues and at the level of the messaging infrastructure.

- Message processing is no longer just moving messages between locations but designers can associate rules and processing steps to be executed when given messages are moved around (e.g., transformations).

- The downside of this approach is that the logic associated with the queues and the messaging middleware might be very difficult to understand since it is distributed and there is no coherent view of the whole.
Publish/Subscribe

- Standard client/server architectures and queuing systems assume the client and the server know each other (through an interface or a queue)

- In many situations, it is more useful to implement systems where the interaction is based on announcing given events:
  - a service publishes messages/events of given 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 the interested clients; this is usually done by transferring the event’s message into a queue for that client

- publish, subscribe, get, .. are also RPC calls to a resource manager
Events

RFQ processing

A: subscription to message *quote*
B: subscription to message *quoteRequest*
C: subscription to message *newQuote*

at systems startup time (can occur in any order, but all must occur before RFQs are executed)

1: publication of a *quoteRequest* message
2: delivery of message *quoteRequest*
3: synchronous invocation of the *getQuote* function
4: publication of a *quote* message
5: delivery of message *quote*
6: publication of a *newQuote* message
7: delivery of message *newQuote*
8: invocation of the *createForecastEntry* procedure

at run time: processing of a request for quote.

SmartQuotation adapter

SmartForecasting adapter
Enterprise Service Bus (ESB)

- An ESB is a message/event oriented architecture that uses messaging as the backbone of the integration
  - The ESB is in charge of routing, publication and subscription, and filtering
  - Applications communicate through the ESB not being aware of what is behind or who is the recipient of what is being sent

- Allows many patterns of communication:
  - Broadcast (sending the same message to many recipients)
  - Mediated transmission (messages are filtered before being forwarded)
  - Can be nested in different scopes

Example: Oracle Service Bus
ESB as a translation step
An integration backbone