Lecture 3: Synchronous Interaction Patterns
Traditional Middleware

Gustavo Alonso
Systems Group
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
alonso@inf.ethz.ch
Reading assignment 2

- RPC and is evolution as middleware

Goal
- Understand how functionality such as RPC evolves into a complex infrastructure supporting application integration.

Assignment:
- Read the following documents:
  - The Rise and Fall of Corba, Michi Henning, ACM Queue, June 2006 (Communications of the ACM, Vol. 51 No. 8, Pages 52-57)
  - The trouble with CORBA, David Chappell
  - CORBA specification
- Write a report (10 pages) discussing how CORBA extends RPC and how much of the functionality in CORBA I related to integration. Compare with the alternative technologies discussed in the course in terms of effectiveness and design options. Discuss how CORBA has influenced existing systems.

- Deadline: 16th November, 18:00 pm => send per e-mail to me (pdf only)
Middleware Platforms
To understand middleware, one needs to understand its dual role as programming abstraction and as infrastructure for integration

<table>
<thead>
<tr>
<th>PROGRAMMING ABSTRACTION</th>
<th>INTEGRATION INFRASTRUCTURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intended to hide low level details of hardware, networks, and distribution</td>
<td>Intended to provide a comprehensive platform for developing and running complex distributed systems</td>
</tr>
<tr>
<td>Trend is towards increasingly more powerful primitives that, without changing the basic concept of RPC, have additional properties or allow more flexibility in the use of the concept</td>
<td>Trend is towards service oriented architectures at a global scale and standardization of interfaces</td>
</tr>
<tr>
<td>Evolution and appearance to the programmer is dictated by the trends in programming languages (RPC and C, CORBA and C++, RMI and Java, Web services and XML)</td>
<td>Another important trend is towards single vendor software stacks to minimize complexity and streamline interaction</td>
</tr>
<tr>
<td>Evolution is towards integration of platforms and flexibility in the configuration (plus autonomic behavior)</td>
<td></td>
</tr>
</tbody>
</table>
Programming abstractions

- Programming languages and almost any form of software system evolve always towards higher levels of abstraction
  - hiding hardware and platform details
  - more powerful primitives and interfaces
  - leaving difficult task to intermediaries (compilers, optimizers, automatic load balancing, automatic data partitioning and allocation, etc.)
  - reducing the number of programming errors
  - reducing the development and maintenance cost of the applications developed by facilitating their portability
Middleware and programming

- Middleware is primarily a set of programming abstractions developed to facilitate the development of complex distributed systems
  - extensions needed to perform middleware specific operations (persistence, transactions, lookup, etc.)
  - to understand a middleware platform one needs to understand its programming model
  - from the programming model the limitations, general performance, and applicability of a given type of middleware can be determined in a first approximation
  - the underlying programming model also determines how the platform will evolve and fare when new technologies appear
And Web Services? And Java?

- Programming abstractions are a key part of middleware but there’s more:
  - a programming abstraction requires a good supporting infrastructure (i.e., a good implementation and support system underneath)

- Programming abstractions, in fact, appear in many cases as a consequence of changes in the underlying hardware or the nature of the systems being integrated

- Java is a programming language that abstracts the underlying hardware: programmers see only the Java Virtual Machine regardless of what hardware platform they use
  - code portability (not the same as code mobility)
  - the first step towards standardizing middleware abstractions (since now the can be based on a virtual platform everybody agrees upon)

- Web services apply existing abstractions to a specialized kind of network: the Internet.
  - The Simple Object Access Protocol (SOAP) of Web services can be seen as a form of RPC wrapped in XML and mapped to HTTP for easy transport through the Internet
The genealogy of middleware

- **Remote Procedure Call**: hides communication details behind a procedure call and helps bridge heterogeneous platforms.

- **Sockets**: operating system level interface to the underlying communication protocols.

- **TCP, UDP**:
  - User Datagram Protocol (UDP) transports data packets without guarantees.
  - Transmission Control Protocol (TCP) verifies correct delivery of data streams.

- **Internet Protocol (IP)**: moves a packet of data from one node to another.

Specialized forms of RPC, typically with additional functionality or properties but almost always running on RPC platforms.
Infrastructure for integration

- As the programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional software layers
  - The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions
  - True even for simple systems (basic RPC)

Middle tier becomes the place to perform the integration
Middleware as infrastructure

DCE runtime environment

IDL

IDL compiler

interface headers

RPC run time service library

RPC protocols

security service

cell service

distributed file service

thread service

DCE runtime environment

client process

DCE development environment

client code

language specific call interface

client stub

RPC API

RPC run time service library

server process

server code

language specific call interface

server stub

RPC API
Evolving middleware

- Middleware evolves and grows as the functionality made available at the middle tier grows and evolves to encompass more options.
- The infrastructure is intended to support additional functionality that makes development, maintenance, and monitoring easier and less costly.
  - RPC => transactional RPC => logging, recovery, advanced transaction models, language primitives for transactional demarcation, transactional file system, etc.
  - The infrastructure is also there to take care of all the non-functional properties typically ignored by data models, programming models, and programming languages: performance, availability, recovery, instrumentation, maintenance, resource management, etc.
Remote Procedure Call
Basic middleware: RPC

- One cannot expect the programmer to implement a complete infrastructure for every distributed application. Instead, one can use an RPC system (our first example of low level middleware).

- What does an RPC system do?
  - Hides distribution behind procedure calls
  - Provides an interface definition language (IDL) to describe the services
  - Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service system
RPC in perspective

**ADVANTAGES**

- 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 fitted quite well with the most typical programming languages (C), thereby facilitating its adoption by system designers.
- RPC allowed the modular and hierarchical design of large distributed systems:
  - client and server are separate
  - the server encapsulates and hides the details of the back end systems (such as databases)

**DISADVANTAGES**

- RPC is not a standard, it is an idea that has been implemented in many different ways (not necessarily compatible).
- RPC allows designers to build distributed systems but does not solve many of the problems distribution creates. In that regard, it is only a low level construct.
- RPC was designed with only one type of interaction in mind: client/server. This reflected the hardware architectures at the time when distribution meant small terminals connected to a mainframe. As hardware and networks evolve, more flexibility was needed.
RPC system issues

- RPC was one of the first tools that allowed the modular design of distributed applications
- RPC implementations tend to be quite efficient in that they do not add too much overhead. However, a remote procedure is always slower than a local procedure:
  - should a remote procedure be transparent (identical to a local procedure)? (yes: easy of use; no: increase programmer awareness)
  - should location be transparent? (yes: flexibility and fault tolerance; no: easier design, less overhead)
  - should there be a centralized name server (binder)?
- RPC can be used to build systems with many layers of abstraction. However, every RPC call implies:
  - Several messages through the network
  - At least one context switch (at the client when it places the call, but there might be more)
  - Threads are typically used in the server to handle concurrent requests
- When a distributed application is complex, deep RPC chains are to be avoided
The Distributed Computing Environment is a standard implementation of RPC and a distributed run-time environment provided by the Open Software Foundation (OSF). It provides:

- RPC
- Cell Directory: A sophisticated Name and Directory Service
- Time: for clock synchronization across all nodes
- Security: secure and authenticated communication
- Distributed File: enables sharing of files across a DCE environment
- Threads: support for threads and multiprocessor architectures
DCE architecture

- **Client process**
  - client code
  - language specific call interface
  - client stub
  - IDL
    - IDL sources
    - IDL compiler
    - interface headers
  - RPC run time service library
  - RPC API
  - security service
  - cell service
  - distributed file service
  - thread service

- **DCE development environment**
  - IDL

- **Server process**
  - server code
  - language specific call interface
  - server stub
  - RPC API
  - RPC run time service library

**DCE runtime environment**
DCE’s model and goals

- Not intended as a final product but as a basic platform to build more sophisticated middleware tools
- Its services are provided as the most basic services needed in any distributed system. Any other functionality needs to be implemented on top of it
- DCE is not just an specification of a standard (e.g., CORBA) but an implementation that acts as the standard. Since the API is the same across all platforms, interoperability is always guaranteed
- DCE is packaged in a modular way so that services that are not used do not need to be licensed

Encina (a TP-Monitor) is an example of an extension of DCE:
On Object Orientation
Efforts like CORBA, OpenDOC or OLE tried to establish a common framework for component based software design. The idea is to treat everything (even entire applications) as an object and provide the means to incorporate these objects into one’s applications.

Implementation of these standards have incorporated much of the functionality found in traditional middleware.

Today, platform like J2EE and .NET reflect this programming language - platform based approach to middleware.

The starting point for these efforts was CORBA.
The Common Object Request Broker Architecture (CORBA) is part of the Object Management Architecture (OMA) standard, a reference architecture for component based systems.

The key parts of CORBA are:

- **Object Request Broker (ORB):** in charge of the interaction between components
- **CORBA services:** standard definitions of system services
- **A standardized IDL language for the publication of interfaces**
- **Protocols for allowing ORBs to talk to each other**

CORBA was an attempt to modernize RPC by making it object oriented and providing a standard.
CORBA follows the RPC model

- CORBA follows the same model as RPC:
  - they are trying to solve the same problem
  - CORBA is often implemented on top of RPC

- Unlike RPC, however, CORBA proposes a complete architecture and identifies parts of the system to much more detail than RPC ever did (RPC is an inter-process communication mechanism, CORBA is a reference architecture that includes an inter-process communication mechanism)

- CORBA standardized component based architectures but many of the concepts behind were already in place long before

- Development is similar to RPC:
  - define the services provided by the server using IDL (define the server object)
  - compile the definition using an IDL compiler. This produces the client stub (proxy, server proxy, proxy object) and the server stub (skeleton). The method signatures (services that can be invoked) are stored in an interface repository
  - Program the client and link it with its stub
  - Program the server and link it with its stub

- Unlike in RPC, the stubs make client and server independent of the operating system and programming language
In order for ORBs to be a truly universal component architecture, there has to be a way to allow ORBs to communicate with each other (one cannot have all components in the world under a single ORB).

For this purpose, CORBA provides a General Inter-ORB Protocol (GIOP) that specifies how to forward calls from one ORB to another and get the requests back.

The Internet Inter-ORB Protocol (IIOP) specifies how GIOP messages are translated into TCP/IP.

There are additional protocols to allow ORBs to communicate with other systems.

The idea was sound but came too late and was soon superseded by Web services.
The best of two worlds: Object Monitors

Middleware technology should be interpreted as different stages of evolution of an “ideal” system. Current systems do not compete with each other per se, they complement each other. The competition arises as the underlying infrastructures converge towards a single platform:

- **OBJECT REQUEST BROKERS (ORBs):**
  Reuse and distribution of components via an standard, object oriented interface and number of services that add semantics to the interaction between components.

- **TRANSACTION PROCESSING MONITORS:**
  An environment to develop components capable of interacting transactionally and the tools necessary to maintain transactional consistency.

What about Object Transaction Monitors?
Object Monitor = ORB + TP-Monitor
On programming languages and EAI

- Today, EAI efforts have detached themselves from the programming language:
  - Legacy applications will always exist
  - The language is not as important as the infrastructure
  - The world is not object oriented, much less in EAI
  - Objects are programming constructs, not integration constructs

- Existing conventional programming languages still carry with them a substantial conceptual legacy from old:
  - Variables as well defined memory regions
  - Interfaces based on parameters, parameters that are variables
  - Function/method call as the basis for interaction
  - Impedance mismatches with integration systems

- Web services and Service Oriented Architectures are trying to resolve exactly these problems (we will study them next)
Transaction Processing monitors
Processing, storing, accessing and retrieving data has always been one of the key aspects of enterprise computing. Most of this data resides in relational database management systems, which have well defined interfaces and provided very clear guarantees to the operations performed over the data.

However:

- not all the data can reside in the same database
- the application is built on top of the database. The guarantees provided by the database need to be understood by the database running on top
The nice thing about databases ...

- … is that they take care of all aspects related to data management, from physical storage to concurrency control and recovery.

- Using a database can reduce the amount of code necessary in a large application by about 40%.

- From a client/server perspective, the databases help in:
  - concurrency control: many servers can be connected in parallel to the same database and the database will still have correct data.
  - recovery: if a server fails in the middle of an operation, the database makes sure this does not affect the data or other servers.

- Unfortunately, these properties are provided only to operations performed within the database. In principle, they do not apply when:
  - An operation spawns several databases.
  - The operations access data not in the database (e.g., in the server).

- To help with this problem, the Distributed Transaction processing Model was created by X/Open (a standard’s body). The heart of this model is the XA interface for 2 Phase Commit, which can be used to ensure that an operation spawning several databases enjoy the same atomicity properties as if it were executed in one database.
What can go wrong here?

- RPC is a point to point protocol in the sense that it supports the interaction between two entities: the client and the server.
- When there are more entities interacting with each other (a client with two servers, a client with a server and the server with a database), RPC treats the calls as independent of each other. However, the calls are not independent.
- Recovering from partial system failures is very complex. For instance, the order was placed but the inventory was not updated, or payment was made but the order was not recorded …
- Avoiding these problems using plain RPC systems is very cumbersome.
2 Phase Commit

**BASIC 2PC**

- Coordinator send PREPARE to all participants.
- Upon receiving a PREPARE message, a participant sends a message with YES or NO (if the vote is NO, the participant aborts the transaction and stops).
- Coordinator collects all votes:
  - All YES = Commit and send COMMIT to all others.
  - Some NO = Abort and send ABORT to all which voted YES.
- A participant receiving COMMIT or ABORT messages from the coordinator decides accordingly and stops.

**What is needed to run 2PC?**

- **Control of Participants**: A transaction may involve many resource managers, somebody has to keep track of which ones have participated in the execution.
- **Preserving Transactional Context**: During a transaction, a participant may be invoked several times on behalf of the same transaction. The resource manager must keep track of calls and be able to identify which ones belong to the same transaction by using a transaction identifier in all invocations.
- **Transactional Protocols**: somebody acting as the coordinator in the 2PC protocol
- Make sure the participants understand the protocol (this is what the XA interface is for)
One at a time interaction

- Databases follow a single thread execution model where a client can only have one outstanding call to one and only one server at any time. The basic idea is one call per process (thread).

- Databases provide no mechanism to bundle together several requests into a single work unit.

- The XA interface solves this problem for databases by providing an interface that supports a 2 Phase Commit protocol. However, without any further support, the client becomes the one responsible for running the protocol which is highly impractical.

- An intermediate layer is needed to run the 2PC protocol.
RPC has the same limitations as a database: it was designed for one at a time interactions between two end points. In practice, this is not enough:

a) the call is executed but the response does not arrive or the client fails. When the client recovers, it has no way of knowing what happened

b) it is not possible to join two calls into a single unit (neither the client nor the server can do this)
Transactional RPC

- The solution to this limitation is to make RPC calls transactional, that is, instead of providing plain RPC, the system should provide TRPC.

- What is TRPC?
  - same concept as RPC plus …
  - additional language constructs and run time support (additional services) to bundle several RPC calls into an atomic unit
  - usually, it also includes an interface to databases for making end-to-end transactions using the XA standard (implementing 2 Phase Commit)
  - and anything else the vendor may find useful (transactional callbacks, high level locking, etc.)

- Simplifying things quite a bit, one can say that, historically, TP-Monitors are RPC based systems with transactional support. An example: Encina.
Transactional RPC

- The limitations of RPC can be resolved by making RPC calls transactional. In practice, this means that they are controlled by a 2PC protocol.
- As before, an intermediate entity is needed to run 2PC (the client and server could do this themselves but it is neither practical nor generic enough).
- This intermediate entity is usually called a transaction manager (TM) and acts as intermediary in all interactions between clients, servers, and resource managers.
- When all the services needed to support RPC, transactional RPC, and additional features are added to the intermediate layer, the result is a TP-Monitor.
Basic TRPC (making calls)

1. **Client**
   - BOT
   - Service_call

2. **Client stub**
   - Get tid from TM
   - Add tid to call

3. **Transaction Manager (TM)**
   - Generate tid
   - Store context for tid

4. **Associate server to tid**

5. **Server stub**
   - Get tid
   - Register with the TM

   **Server**
   - Invoke service
   - Return
Basic TRPC (committing calls)

Client
... Service_call ...
EOT

Client stub
Send to TM commit(tid)

Transaction Manager (TM)
Look up tid

Run 2PC with all servers associated with tid

Server stub
Participant in 2PC

Server
The previous example assumes the server is transactional and can run 2PC. This could be, for instance, a stored procedure interface within a database. However, this is not the usual model.

Typically, the server invokes a resource manager (e.g., a database) that is the one actually running the transaction.

This makes the interaction more complicated as it adds more participants but the basic concept is the same:
- The server registers the resource manager(s) it uses.
- The TM runs 2PC with those resource managers instead of with the server (see OTS at the end).
TP-Monitors = transactional RPC

- A TP-Monitor allows building a common interface to several applications while maintaining or adding transactional properties. Examples: CICS, Tuxedo, Encina.

- A TP-Monitor extends the transactional capabilities of a database beyond the database domain. It provides the mechanisms and tools necessary to build applications in which transactional guarantees are provided.

- TP-Monitors are, perhaps, the best, oldest, and most complex example of middleware. Some even try to act as distributed operating systems providing file systems, communications, security controls, etc.

- TP-Monitors have traditionally been associated to the mainframe world. Their functionality, however, has long since migrated to other environments and has been incorporated into most middleware tools.
TP-Monitor functionality I

- TP-Monitors appeared because operating systems are not suited for transactional processing. TP-Monitors are built as operating systems on top of operating systems.

- As a result, TP-Monitor functionality is not well defined and very much system dependent.

- A TP-Monitor tries to cover the deficiencies of existing “all purpose” systems. What it does is determined by the systems it tries to ”improve”.

- A TP-Monitor is basically an integration tool. It allows system designers to tie together heterogeneous system components using a number of utilities that can be mixed and matched depending on the particular characteristics of each case.

- Using the tools provided by the TP-Monitor, the integration effort becomes more straightforward as most of the needed functionality is directly supported by the TP-Monitor.
TP-Monitor functionality II

- A TP-Monitor addresses the problems of sharing data from heterogeneous, distributed sources, providing clean interfaces and ensuring ACID properties.

- A TP-Monitor extrapolates the functions of a transaction manager (locking, scheduling, logging, recovery) and controls the distributed execution. As such, TP-Monitor functionality is at the core of the integration efforts of many software producers (databases, workflow systems, CORBA providers, …).

- A TP-Monitor also controls and manages distributed computations. It performs load balancing, monitoring of components, starting and finishing components as needed, routing of requests, recovery of components, logging of all operations, assignment of priorities, scheduling, etc. In many cases it has its own transactional file system, becoming almost indistinguishable from a distributed operating system.
Transactional properties

- The TP-monitor tries to encapsulate the services provided within transactional brackets. This implies RPC augmented with:
  - **Atomicity**: a service that produces modifications in several components should be executed entirely and correctly in each component or should not be executed at all (in any of the components).
  - **Isolation**: if several clients request the same service at the same time and access the same data, the overall result will be as if they were alone in the system.
  - **Consistency**: a service is correct when executed in its entirety (it does not introduce false or incorrect data into the component databases).
  - **Durability**: the system keeps track of what has been done and is capable of redoing and undoing changes in case of failures.
# include <tc/tc.h>
inModule("helloWorld");

void Main () {
    int i;
    inFunction("main");
    initTC(); */ initializes transaction manager */

    transaction { /* starts a transaction */
        printf("Hello World - transaction %d\n", getTid());
        if (I % 2) abort ("Odd transactions are aborted");
    }
    onCommit
        printf("Transaction Comitted");
    onAbort
        printf("Abort in module: %s\n \t %s\n", abortModuleName(), abortReason());
}
TP-Monitor, generic architecture

Yearly balance?

Monthly average revenue?

Front end

TP-Monitor environment

Control (load balancing, cc and rec., replication, distribution, scheduling, priorities, monitoring …)

Interfaces to user defined services

Programs implementing the services

app server 1

app server 2

app server 3

recoverable queue

wrappers

Branch 1

Branch 2

Finance Dept.
Tasks of a TP Monitor

Core services

- Transactional RPC: Implements RPC and enforces transactional semantics, scheduling operations accordingly
- Transaction manager: runs 2PC and takes care of recovery operations
- Log manager: records all changes done by transactions so that a consistent version of the system can be reconstructed in case of failures
- Lock manager: a generic mechanism to regulate access to shared data outside the resource managers

Additional services

- Server monitoring and administration: starting, stopping and monitoring servers; load balancing
- Authentication and authorization: checking that a user can invoke a given service from a given terminal, at a given time, on a given object and with a given set of parameters (the OS does not do this)
- Data storage: in the form of a transactional file system
- Transactional queues: for asynchronous interaction between components
- Booting, system recovery, and other administrative chores
Structure of TP-Monitors (I)

- TP-Monitors try in many aspects to replace the operating system so as to provide more efficient transactional properties. Depending what type of operating system they try to replace, they have a different structure:
  - **Monolithic**: all the functionality of the TP-Monitor is implemented within one single process. The design is simpler (the process can control everything) but restrictive (bottleneck, single point of failure, must support all possible protocols in one single place).
  - **Layered**: the functionality is divided in two layers. One for terminal handling and several processes for interaction with the resource managers. The design is still simple but provides better performance and resilience.
  - **Multiprocessor**: the functionality is divided among many independent, distributed processes.
Structure of TP-Monitors (II)

Terminal handling (multithreaded)

Layered structure

Multiprocessor structure
TP-Monitor components (generic)

From “Transaction Processing” Gray&Reuter. Morgan Kaufmann 1993
Example: BEA Tuxedo

- **Service Call**
  - Client dll
  - Process routine
  - Service call

- **Forward Call**
  - Locate server
  - Server location

- **Queue**
  - Forward call
  - Read
  - Invoke

- **Transaction**
  - Response

- **Bulletin Board**
  - Client handler
  - Name server

- **Server Process**
  - Named service

- **Resource Manager**
Example: BEA Tuxedo

- The client uses DLL (Dynamic Link Libraries) routines to interact with the TP-Monitor.

- The Monitor Process or Tuxedo server implements all system services (name services, transaction management, load balancing, etc) and acts as the control point for all interactions.

- Application services are known as named services. These named services interact with the system through a local server process.

- Interaction across components is through message queues rather than direct calls (although clients and servers may interact synchronously).
TP-Heavy vs. TP-Light = 2 tier vs. 3 tier

- A TP-heavy monitor provides:
  - a full development environment (programming tools, services, libraries, etc.),
  - additional services (persistent queues, communication tools, transactional services, priority scheduling, buffering),
  - support for authentication (of users and access rights to different services),
  - its own solutions for communication, replication, load balancing, storage management ... (similar to an operating system).

- Its main purpose is to provide an execution environment for resource managers (applications), with guaranteed reasonable performance

- This is the traditional monitor: CICS, Encina, Tuxedo.

- A TP-Light is a database extension:
  - it is implemented as threads, instead of processes,
  - it is based on stored procedures ("methods" stored in the database that perform an specific set of operations) and triggers,
  - it does not provide a development environment.

- Light Monitors are appearing as databases become more sophisticated and provide more services, such as integrating part of the functionality of a TP-Monitor within the database.

- Instead of writing a complex query, the query is implemented as a stored procedure. A client, instead of running the query, invokes the stored procedure.

- Stored procedure languages: Sybase's Transact-SQL, Oracle's PL/SQL.
Databases and the 2 tier approach

- Databases are traditionally used to manage data.
- However, simply managing data is not an end in itself. One manages data because it has some concrete application logic in mind. This is often forgotten when considering databases.
- But if the application logic is what matters, why not move the application logic into the database? These is what many vendors are advocating. By doing this, they propose a 2 tier model with the database providing the tools necessary to implement complex application logic.
- These tools include triggers, replication, stored procedures, queuing systems, standard access interfaces (ODBC, JDBC).
Advantages of TP-Monitors

- TP-Monitors are a development and run-time platform for distributed applications.

- The separation between the monitor and the transaction manager was a practical consideration but turned out to be a significant advantage as many of the features provided by the monitor are as valuable as transactions.

- The move towards more modular architectures prepared TP-Monitors for changes that had not been foreseen but turned be quite advantageous:
  - The web as the main interface to applications: the presentation services included an interface so that requests could be channeled through a web server.
  - Queuing as a form of middleware in itself (Message Oriented Middleware, MOM): once the queuing service was an internal resource manager, it was not too difficult to adapt the interface so that the TP-Monitor could talk with other queuing systems.
  - Distributed object systems (e.g., CORBA) required only a small syntactic layer in the development tools and the presentation services so that services will appear as objects and TRPC would be come a method invocation to those objects.
Back to objects
Object Transaction Service

- An OTS provides transactional guarantees to the execution of invocations between different components of a distributed application built on top of an ORB. It is part of the CORBA standard. It is identical to a basic TP-Monitor.

- There are two ways to trace calls:
  - Explicit (manual): the invocation itself contains the transaction identifier. Then, when the application registers the resource manager, it uses this transaction identifier to say to which transaction it is "subscribing."
  - Implicit (automatic): the call is made through the OTS, which will forward the transaction identifier along with the invocation. This requires to link with the OTS library and to make all methods involved transactional.

- ... and two ways to register resources (necessary in order to tell the OTS who will participate in the 2PC protocol and what type of interface is supported)

- Manual registration implies the user provides an implementation of the resource. This implementation acts as an intermediary between the OTS and the actual resource manager (useful for legacy applications that need to be wrapped).

- Automatic registration is used when the resource manager understands transactions (i.e., it is a database), in which case it will support the XA interface for 2PC directly. A resource are registered only once, and implicit propagation is used to check which transactions go there.
Running a distributed transaction (1)

1) Assume App A wants to update databases A and B

2) App A obtains a txn identifier for the operation

3) App A registers the database for that transaction

4) App A runs the txn but does not commit at the end
Running a distributed transaction (2)

5) App A now calls App B

6) App B registers the database for that transaction

7) App B runs the txn but does not commit at the end

2) App A request commit and the OTS runs 2PC