Three-Tier Distribution Architecture

Robert Hirschfeld

hirschfeld@rz.tu-ilmenau.de

30. July 1996

Abstract

When designing distributed applications for enterprise information systems you have to decide how to distribute application functionality or responsibility between distributed processing contexts in order to optimize the usage of components and resources involved. The three-tier distribution architecture describes the partitioning of your application functionality into three tiers: front-end clients, domain or application servers, and a storage and DB-server.

Name

Three-Tier Distribution Architecture

Aliases

Three-Tier Client/Server Architecture

Balanced Client/Server Architecture

Context

You have to design applications for distributed enterprise information systems where usually some desktop components will access or modify shared resources, mostly located within a non-active database.

Problem

How can you partition application functionality or responsibility between distributed processing contexts best in order to optimize the usage of components and resources involved?

Forces

You can locate most of the application code on a client. The client has access to a database server that is shared with all other clients using the vendor-supplied API. Even if this architecture is very simple and easy to build, this leads to the so called fat-client problem, addressed by most of the following forces.

When putting most or all application code on the clients, they (the clients) have to request or download all the data they need to fulfill their business tasks. This is sometimes too inefficient. The network becomes overloaded when many clients request a great amount of data (or objects).

Also if a client do not has the chance to delegate the processing of the requested data to another client, application performance or response time depends heavily on the power/efficiency of the system supporting this client.

If you rely on a vendor specific server API, you can estimate the effort to integrate that kind of server. On the other hand you will become too inflexible or fix at all, because it restricts you to a specific set of services the server places to the clients' disposal. If you are trying to prevent getting „locked into“ a specific vendor once you build on top of their proprietary API, you can’t use the built-in features of your database server dealing with concurrency control, transactional behavior and recovery directly. So you have to ensure consistency and availability of your application system on your own.

Using existing interfaces is constraining, but defining your own interfaces takes effort and is error prone.

Distributing the application code between clients and servers makes it more flexible and scaleable, but the larger the number of parts, the more complex a system seems, and the more failure points exist. Therefore the system becomes more difficult to maintain, too.

If your clients interact using a (non-active) database server, automatic change propagation is unsupported, or at best supported by highly vendor specific database triggers mechanisms. This may require continous checking (polling) of the server to detect changes, resulting in extra network load. Depending of the polling time-frame some clients become inconsistent with the database for a while or may be affected by network traffic problems.

Solution

Partition your application functionality or responsibility into three tiers:

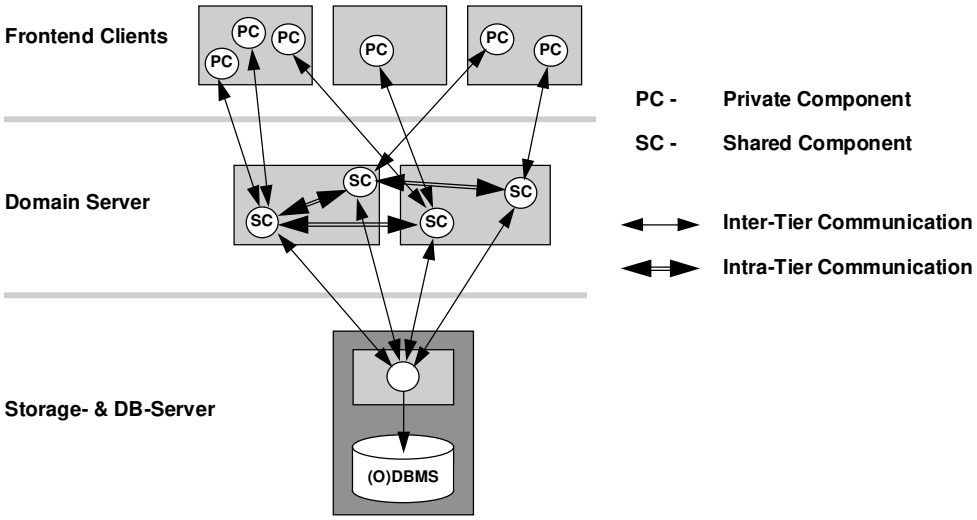
- front-end clients,
- domain servers (also known as application servers),
- and a storage and DB-server.

This results in the so-called three-tier distribution architecture.

The *front-end clients tier* consists of components unique to every user including user specific application logic and the user interface. These single user systems handle the bulk of user interaction and task specific local operations originated by the user.

The *domain server tier*, supported by a multi-user environment, holds the shared parts of application and business logic. This tier needs services like transactions, concurrency control and security. Because most of the data/information requested by the applications are accessed via this layer, data intensive computing operations should be executed here whenever possible. Domain servers are responsible for coordinating transactions and accessing legacy systems. Within this tier parts cooperate via intra-tier communication.

The task of the *storage- and DB-server tier* is to manage persistency of certain data/information and to execute the database transactions. An optional storage server may be responsible for special mapping, e.g. the mapping between objects from the application and tables of a RDBMS or between object-ids within the execution space and object-ids within an ODBMS.



Application processing, located on domain server systems, is separated from both the user interface and the database server. Activities unique to a particular user can be executed local/private. Centralized data can be accessed without moving it completely to the front-end clients.

The possibility of information preprocessing on domain servers helps reduce network traffic to clients and therefore increases performance.

Most of the application code is isolated from user specific interaction and interfaces. All application logic common to some users/applications is maintained in one (single/shared) location. So application code/logic can be changed in the middle tier without affecting front-ends and database server.

The interface between front-ends and domain servers can be designed by the programmer as needed because the programmer doesn't have to fit a vendor-supplied API.

Domain servers provide good possibilities for integrating legacy systems, because the interfaces and bridges to such systems are concentrated and managed in one location for all applications needing access.

Consequences

Even if there are many benefits and advantages of a three-tier distribution architecture, the more freedom and increased flexibility leave the developers with more responsibilities. They have to design interfaces between front-end clients and domain servers and to ensure application consistency and availability themselves.

Nevertheless one possibility to overcome these obstacles is the consequent use of industry standards like CORBA and CORBAservices from the Object Management Group ([OMG94, OMG95, OMG96], see Implementation section).

Implementation

Communication. If your application is intended to run within a distributed heterogeneous environment and to interoperate with other applications of this environment, you should decide to apply standards like CORBA and CORBAservices from the Object Management Group (OMG) as much as possible ([OMG94, OMG95, OMG96]). For an asynchronous decoupled communication the Event Notification Service will fit ([OMG96]).

Interfaces. You have to describe the interface between your application parts for inter- and intra-tier communication/interaction. In order to stay or become platform and language independent, you can use a descriptive language like IDL (Interface Definition Language) from the CORBA specification of the OMG ([OMG95]).

Sharing & Concurrency. Some of your application parts share certain resources outside the scope of a database. So you are responsible for coordinating concurrent access to these shared resources in order to ensure their consistency. Here you can apply the OMG Concurrency Control Service to serialize access ([OMG96]).

Reliability. While sharing resources outside a database, you want or have to ensure the consistency of all of them as a group or unit. To achieve this you can introduce transactional behavior into your whole application system. One possible approach may be the Object Transaction Service of the OMG ([OMG96]).

Initial Connections/References. If a private component of an application wants to establish an initial connection to a certain shared component, it has to look for it. Here the Naming Service may help by supporting you to reach/access an object by a symbolic/meaningful name ([OMG96]).

Acknowledgments

Thanks are due to Gerard Meszaros for reviewing this paper and for giving helpful hints for improvement.

References

[OMG94] *Object Services Architecture.*
Object Management Group, December 1994

[OMG95] *The Common Object Request Broker: Architecture and Specification*
Object Management Group, July 1995

[OMG96] *CORBAservices: Common Object Services Specification.*
Object Management Group, March 1996