A Command and Control Support System Using CORBA

José A. Rodrigues Nt.
Vera M. Ulm de G. Lima
Gladys M. P. S. Lima
Marcelo Corsino Ferreira
José Carlos Alves de Almeida
CASNAV - Brazilian Navy
rneto@attglobal.net, ulm@casnav.mar.mil.br,
{gladys, corsino, jcarlos} @casnav.mar.mil.br

Sylvia de Oliveira e Cruz
Renato F. G. Cerqueira
Carlos Bazílio Martins
Departamento de Informática
Pontifícia Universidade Católica
{svi, rcerq, bazilio}@inf.puc-rio.br

Abstract

A C4I (Command, Control, Computers, Communication and Intelligence) support system spans a large variety of requirements and usually serves many users with diverse needs. Also, in order to properly and timely display information to the decision-makers, it shall integrate data from other systems, not necessarily built with the same technology.

The Operations Theater Surveillance System (SATO), presented here, was designed in 1998 for the Mercury Project, which develops the Brazilian Navy new Command and Control support system (C2S). This paper, as an experience report, focuses on the software constructs used in the system, with an architectural perspective. SATO displays and manages all the information presented to the users. In addition, as the Mercury’s main subsystem, it lays the foundation for integrating other subsystems and legacy systems. Data enters Mercury from all Crisis Control Centers (CCC), distributed throughout the country.

Keywords: Distributed Systems, C4I, C2S, Command and Control, CORBA, Software Architecture.

1. Introduction

The Mercury Project develops an integrated C4I support system. This paper presents a solution to face the many diverse and demanding requirements established for the system, showing the software architecture created and explaining its main features.

The Project’s main system is the Operations Theater Surveillance System – SATO. This system provides the user an on-line graphical presentation of all resources being controlled/surveilled and their cinematic, overlaid on a geographical map background. It is also responsible for integrating data from other systems. Data comes from all Crisis Control Centers (CCC), distributed throughout the country, both from other instances of SATO running on these locations and from feeds of other systems.

2. The Problem

Displaying synchronized information on all CCC’s, while being a platform for integrating information originating from different sources, was the basic problem for SATO. Many other requirements, many very similar to the ones described by Roodyn [1] existed:

1. Capable of being integrated to other systems, taking differing types of data;
2. Accept data from more than one source. Data synchronization has to be accomplished;
3. Allow the integration of selected Decision Support Systems;
4. Provide a way to derive data, including forecasting, mostly on the clients, using the parameters provided by the interested user;
5. Clients should be able to select data to be presented and how it is presented;
6. Server should be able to filter data to be transmitted based on client type, classification and other criteria established by the user;
7. Clients should be notified of modifications in a timely manner;
8. Store data for historical analysis and auditing; and
9. Be reliable, supporting uninterrupted operation 24x7x365, in any condition, including network failure.

It was also required that the system should be built on Windows NT, using both Digital Alpha and Intel
machines, with the Microsoft SQL Server as the system’s DBMS.

3. The Solution

Requirement 1, 2 and 3 called for a solution based on a middleware capable of creating some type of abstraction layer to facilitate integration. Besides being a standard, CORBA would provide several advantages, such as the use of multithreading on both client and server programs and reduction of the complexity of the server-side software [2]. IONA Orbix was selected due to its stability and multiple platform and language support [2] [3].

Although strict time constraints were not explicit, the nature of the system required a real-time approach. This way, and meeting requirement 7, the solution should be able to provide a soft real-time environment [4] [5].

Requirements 4 and 5 would ask for client applications allowing different scenarios and simulations to be built by each user, and even different clients, custom-built for the users’ needs. Also taking into account requirement 8, a layered architecture would provide several benefits on this case [6].

An Event-based architectural style would help in implementing requirements 6 and 7 [6], and also 1 and 3 [7] [8]. Finally, requirement 9 demanded the object distribution to be done in a way that operation would continue even in the event of a network failure.

In view of the strategic nature of the system and that it would run on dedicated hardware, and integrated with legacy code, a distributed objects architecture was chosen as the basic solution [9].

The basis for building the solution was the MVC concept [10]. The Trim and Fit Client pattern [11] was then used as the starting point to design the system. Figure 1 shows the basic architectural construction.

4. Implementing the Solution

SATO’s development was divided in three main phases:

- 1st Phase - Stand-alone;
- 2nd Phase - Distributed Map-based presentation based on centralized objects’ server/databases; and
- 3rd Phase - Distributed and replicated objects’ servers/databases (local to each CCC).

In its second stage, as a three-tier system, SATO relies on centralized objects’ server/databases located in one of the CCC’s. It displays all the information required by the C4I structure. This means, for instance, that besides displaying the target’s position and movement, it also displays detailed data about the target, upon request. The client applications access the server application – the object manager – through a network distributed all around the country.

SATO is designed as a four-layer system (figure 1) [11]. The two top layers reside on the clients and are in the user-working environment. Each different client application has the same basic building blocks, except for the visual objects (VO), used only on map-based applications.

The VO’s were conceived to avoid unnecessary use of memory and performance degradation. They are special copies of the objects that reside on the server — domain objects (DO) — that remain in memory during the whole application execution. They keep the essential information existent on their domain counterpart and some data needed for their presentation, like its position on the screen. Since the VO’s are dynamic, its movements are calculated locally, on the client. In addition, since the users constantly access part of the basic data about targets, like target’s classification, course and speed, these data is kept on the VO’s too. This characteristic also presents an extra benefit — the possibility of implementing simple mechanisms that allow the targets’ presentations to be customized on a per client basis.

The Application Manager (AppM) is typical of each client application, i.e., each client type has to have its specific AppM. The two main clients in the system now are the Graphical Presentation Module (GPM – map-based) and the Data Management Module (DMM). In the Map-based Clients, for instance, AppM is responsible for keeping track of collisions between VO’s.

The lower two layers reside on the server. The domain layer constitutes the main part of the Distributed Objects’ Server Module (DOSM). It is responsible for data distribution and synchronization. It also provides services to other applications that communicate with the system.

The Infrastructure layer is where the persistence of the DO’s is done. It uses SQL, through named pipes.

As a C2S, any new information introduced in any of the systems’ workstations shall be immediately notified to all others. This characteristic presents a few problems. Two of them are of fundamental importance. First, the system...
deeply relies on networked communications that are not fail safe. Second, it may exist concurrent attempts to update data. To address the former, a mechanism that keeps track of the client's connections "health" is implemented. To the latter, a concurrency control solution was implemented.

As explained earlier, there were no explicit time constraints formally imposed on the system. However, the users constantly evaluate the system and report any problem to the development team, as in [5].

4.1 The Distributed Objects’ Server Module

The DOSM is SATO’s main component. It takes care of all objects distribution, including client’s initialization and object changes multicasting, concurrency and persistence. For the latter, it relies on two databases. The Target Control Database (TCD), which stores all data related to targets’ movements and operations. DOSM is responsible for keeping data synchronized among all client applications and TCD. The other one, Target Information Database (TID), stores detailed information about ships, aircrafts and troops. Both databases are represented in figure 2 as DB. DOSM has no responsibility over the data that exists on the latter, regarding distribution and synchronization — it just retrieves what is requested by the client application, working as a communication link between client applications and TID.

DOSM has two managers: the Domain Manager (DM) and the Event Manager (EM) (Figure 2). The DM is mostly responsible for keeping the DO’s in memory, talking to the databases and keeping them updated. The EM takes care of the DO's distribution. It provides the data necessary for initializing the clients, receives their notifications about changes on DO’s, keeps them informed of every change made by other clients and keeps track of their statuses.

4.2 The Application Manager

On the client side, the Application Manager (AppM) does the essential work. Whenever a client is started, it establishes connection with the server, telling the server that it is an active client, and in return, should be notified of every modification on the DO’s, that are of its interest. The AppM has an interface, called Notification Manager (NoM), which receives, and processes, all notifications from DOSM. Since the number of such events is very large, it uses a separate thread to each. AppM has another significant function: the notifications’ acknowledgements. If, for any reason, the acknowledgement is not received by DOSM, after a certain time it unsubscribes the client.

4.3 The Event Manager

The EM is implemented as a Singleton [12]. It keeps an event queue for each subscribed client. Clients can receive 4 types of events: DOSM objects (Initialization), New Object, Object Changed and Object Removed.

The client’s subscription process starts with a bind on the EM. The client subscribes DOSM services using EM’s proxy. According to the client type, the EM creates a concrete Client. The notifications follow the Distributed Callback pattern [13], avoiding the need of subsequent binds. When the subscription is finished, the client waits for notifications. The DOSM creates an InitializationEvent then. Only after this notification is finished, the client is able to accept regular notifications. A separate thread works for each client, and priority is adjusted to avoid starvation.

Although implemented this way now, the EM was not planned like this. On its first implementation, the subscription data was returned as out parameters of the method, instead of as a separate event created by the DOSM. As originally implemented, it turned out to be ineffective. The preparation of the client’s initialization data is costly, and letting its control on the client did not work. DOSM was eventually interrupted during an important task, staying too long answering a new client’s connection.

4.4 Fault Tolerance

System vulnerability resides on the DOSM — EM, since it is responsible for all communication with the clients. Three basic problems arise:
• Dependency on one single instance, aggravated by the fact that it operates on a network;
• Performance, since every client will depend on it, and the number of clients is not limited; and
• Data integrity, as network disruptions could produce a non-trustable client.

While the first problem will be addressed on the third version of the system, the second and the third were already taken care on this version. In order to reduce the bottleneck, EM was implemented as a multithreaded object, with fault and deadlock detection algorithms.

Typically, EM controls one notification thread per client. These threads can share objects. Semaphores are used to guarantee exclusive access and locks are time-limited to avoid deadlocks.

Besides that, a client’s fault detection algorithm was implemented, so EM would not waste time trying to notify clients that lost their connection, what could end up flooding DOSM. This algorithm defines four states for a client, as shown in figure 3.

The limits (M and N) on figure 3, were determined on system’s trials and can be adjusted to specific network configurations, as this proved necessary.

5. Summary and Future Work

The solution adopted was a mix of several architectural styles. Initially, the performance was not what the users expected but, after improving the subscription process, all fell into place. The Event service [14] implementations tested (ORBIX and COOL Chorus), at the time, were not satisfactory. This choice was also made because we believed that a more flexible filtering mechanism would be required [8] and special adjustments would have to be made for the final version of the system.

Implementing the EM and adjusting its behavior to the network, was a tough job. Changing the way EM worked and providing it the capacity to be adjusted to different networks was crucial to the success. However, phase 3 will be much harder. Requirement 9 imposes synchronization of multiple Objects’ Servers and Databases. Presently, the team works on an architecture that will let the clients connect to any DOSM, choosing it dynamically and being able to switch the DOSM being used at any time.

6. Acknowledgements

The expertise of Jose Gomes from the Brazilian Navy Research Lab (IPqM) was of great value. The Project would not have gotten this far without the strong support and smart management of Captain Lander Silva. Their contribution is gratefully acknowledged.

7. References


Figure 3 – EM Client’s State Transition Diagram