A Distributed Command and Control Support System

Track 6

C2 Assessment Tools and Metrics

José A. Rodrigues Nt.
Ivana Cardial de Miranda
Lander Loureiro da Silva
CASNAV
Brazilian Navy
Ilha das Cobras, s/n – AMRJ ed. 8
Rio de Janeiro, RJ 20091 Brazil
+5521 3849-6464
rneto@computer.org,
{ivana, lander}@casnav.mar.mil.br

Sylvia de Oliveira e Cruz
Renato F. G. Cerqueira
Departamento de Informática
Pontifícia Universidade Católica
R. Marquês de São Vicente, 225
Rio de Janeiro, RJ 22451-041 Brazil
+5521 529-9460
{sylvia, rcerq }@inf.puc-rio.br

A Distributed Command and Control Support System

José A. Rodrigues Nt.
Ivana Cardial de Miranda
Lander Loureiro da Silva
CASNAV
Brazilian Navy
Ilha das Cobras, s/n – AMRJ ed. 8
Rio de Janeiro, RJ 20091 Brazil
+5521 3849-6464
rneto@attglobal.net ,
{ivana, lander}@casnav.mar.mil.br

Sylvia de Oliveira e Cruz
Renato F. G. Cerqueira
Departamento de Informática
Pontifícia Universidade Católica
R. Marquês de São Vicente, 225
Rio de Janeiro, RJ 22451-041 Brazil
+5521 529-9460
{sylvia, rcerq }@inf.puc-rio.br

ABSTRACT

The construction of a C^4I (Command, Control, Computers, Communication and Intelligence) support system requires an extreme care with architecture. By its nature, a C^4I system spans a large variety of requirements and usually serves many users with different needs. Also, in order to properly and timely display information to the decision-makers, it shall integrate data from other systems, not necessarily built on the same technology.

The system presented here was designed in 1998 for the Mercury Project, which develops the Brazilian Navy new C^4I support system. This paper, as an experience report, focuses on the main constructs and projects' decisions taken during the development of the Operations Theater Surveillance System (Sistema de Acompanhamento do Teatro de Operações – SATO). SATO displays and manages all the information that is presented to the users. In addition, as the system's main subsystem, it lays the foundation for integrating other subsystems and legacy systems.

Starting with an overview of the whole system, the architecture, the modules and their main behavior are described. Then some considerations are made on the evolutions required and yet to be implemented.

This system provides the user an on-line graphical presentation of all resources being controlled/surveilled and their cinematic, on a geographical map background, as a Geographic Information System.

Data enters Mercury from all Crisis Control Centers (CCC), distributed throughout the country. The information displayed has to be synchronized among all centers, i.e., all decision-makers must work on the same operational picture. Since the system is built for crisis control, whenever a situation develops the information flow speed and reliability are critical. Therefore, a distributed object-oriented architecture was developed to address these issues, using CORBA, the Object Management Group (OMG) standard for object distribution. The paper focuses on the software constructs used in the system, with an architectural perspective, not the C^4I architecture, presenting a successful architecture that can also be used in other distributed systems with similar applications and requirements.

Keywords

Distributed Systems, C^4I, C^2S, Command and Control, CORBA, Architecture.

1 INTRODUCTION

The Mercury Project develops an integrated C^4I support system. As an experience report, this paper presents a solution derived to face the many diverse and demanding requirements established for the system, showing the software architecture created and explaining its main features.

The main subsystem is the Operations Theater Surveillance System – SATO. This subsystem 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 subsystems. Data enters Mercury 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 that should provide data to help the Command and Control (C^2) organization [ROD01].

2 THE PROBLEM

Displaying synchronized information among all centers, i.e., providing all decision-makers with the same operational picture while being a platform for integrating information originating from different sources was the basic problem for SATO. Many other requirements, some very similar to the ones described by Roodyn [ROO99] 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, specially when different sources provide data about the same object;
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 the 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 new data or modifications on existing data, in a timely manner;
8. Store data for historical analysis and auditing purposes. All data changes must be tracked to the user that originated them; and
9. Be reliable, supporting uninterrupted operation 24x7x365, in any condition, including WAN failure.

Due to the fact that the customer was already using Windows NT, it was required that the system should be built on it, 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, avoiding a creation of a proprietary solution with the use of sockets, 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* [URB99]. IONA Orbix was selected due to its stability and multiple platform and language support [URB99] [IONA].

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 [PRESS] [EMM99].

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 [SHAWM].

An Event-based architectural style would help in implementing requirements 6 and 7 [SHAWM], and also 1 and 3 [CAR99] [CUG98]. Finally, requirement 9 demanded the object distribution to be done in a way that operation would continue even in the event of a failure of the network infrastructure.

In view of the strategic nature of the system, since it would run on dedicated hardware and integrated with legacy code, a distributed objects architecture was chosen as the basic solution [SCH99].

Considering all the requirements above, the following characteristics should be part of the adopted architecture:

- Distributed Objects;
- Layered System;
- Event-Based;
- Redundant; and
- Soft Real-Time.

The basis for building the solution was the MVC concept [KRA88]. The Trim and Fit Client pattern [BRO97] was then used as the starting point to design the system.

4 IMPLEMENTING THE SOLUTION

SATO's development was planned to be done in three main phases, which would produce systems with the following characteristics:

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

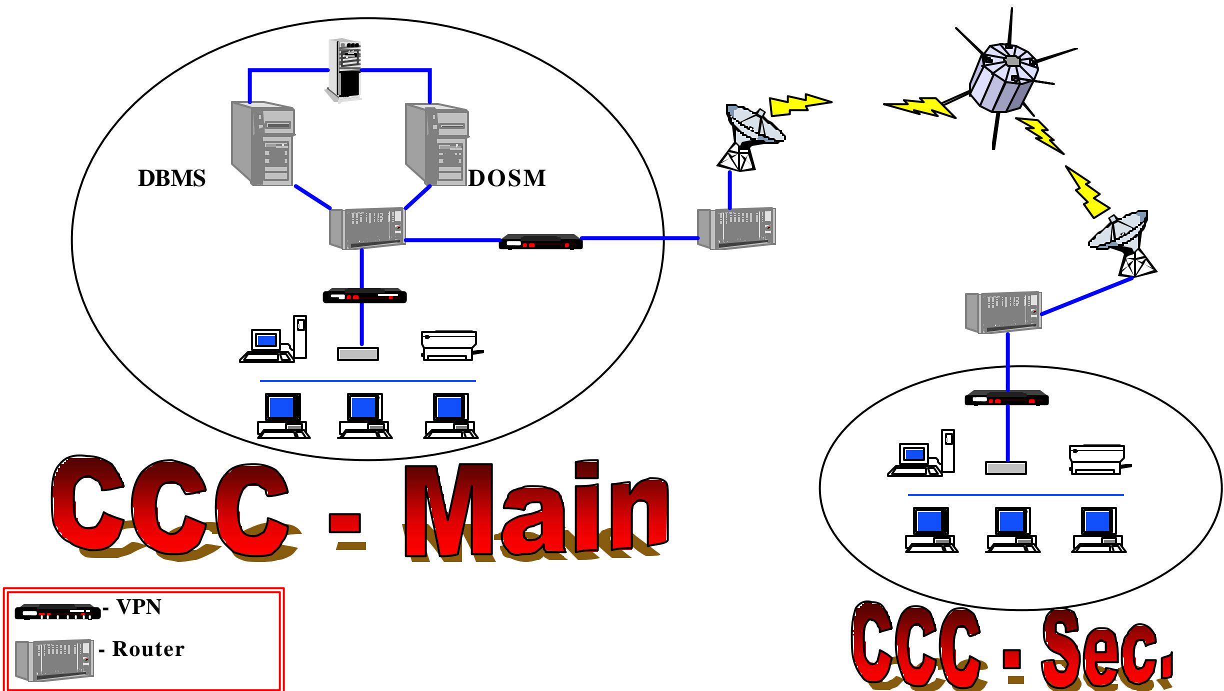

Figure 1 - Mercury Deployment Scheme (2nd phase)

In its second stage, as a three-tier system, SATO relies on centralized objects' server/databases located in one of the CCC's (figure 1). 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. Many client applications access the server side application – the object manager – through a network distributed all around the country. The basic data required by the client applications is kept in memory, on the application server, and persisted in a relational database.

SATO is designed as a four-layer system (figure 2) [BRO97]. The two top layers reside on the dients and are in the user-working environment. Each different client application that composes the system has its basic building blocks, except for the visual objects (VO), just used on map-based applications, i.e., applications that show maps and the targets movements on it.

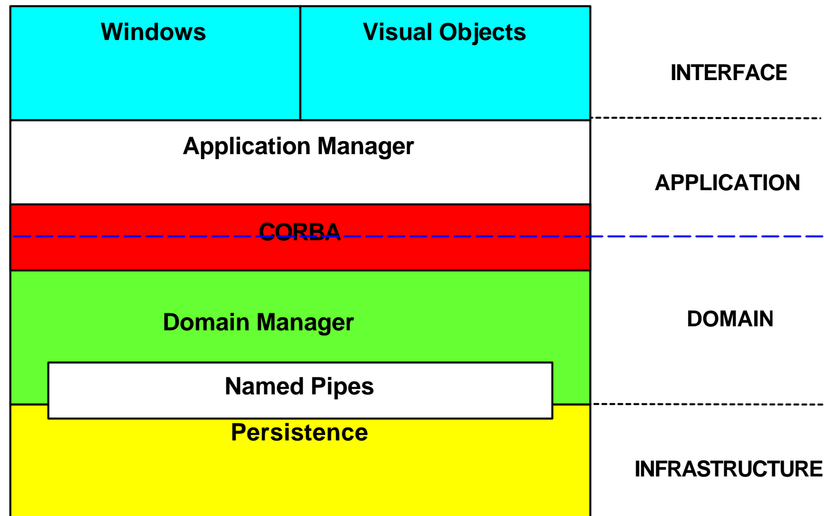

Figure 2 - SATO layered architecture

The VO's were derived to avoid unnecessary use of memory and performance degradation. The VO's 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, i.e., are targets that keep moving the whole time, its movements are calculated locally, on the client, without any need for communication with the servers. 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 feature also presents an extra benefit, which is the possibility of implementing simple mechanisms that allow the presentation of the targets to be customized on a per client basis.

The Application Manager (AppM) is typical of each client application, i.e. each client type has its own AppM. It is responsible for implementing its main functionalities. 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, like the Message Management System (Sistema de Gerência de Comunicações – SGC), which automatically delivers and extracts data from SATO, through a CORBA based, customized API.

The Infrastructure layer is where the persistence of the DO's is done. It uses SQL, through named pipes. The DO's are kept in memory the whole time and changes on its data are written on the databases.

As a C4I support system, any new information introduced in any of the systems' client workstations shall be immediately notified to all others, allowing the operators and decision-makers to assess the changes almost at the same time it was introduced. As explained earlier, there were no explicit time constraints imposed to the system. However, the users evaluate the system constantly and report any problem to the development team as in Roodyn and Emmerich work [EMM99]. 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, since it may exist concurrent attempts to update systems' data, due to sensor information produced by different controllers – CCC, special care is required in a distributed application of this kind. To address the former, Mercury, on its current phase, implements a mechanism that keeps track of the client's connections "health" (discussed on the Fault Tolerance subsection). To the latter, it implements a concurrency control solution.

The Distributed Objects' Server Module

The DOSM is the main server-side component of SATO. It takes care of all objects distribution, including client's initialization and object changes broadcasting, concurrency and persistence. For the latter, it relies on two databases. The first one, Target Control Database (TCD) stores all the information related to targets' movements and operations. It is the back-end for SATO's object persistence. DOSM is responsible for keeping these data synchronized among all client applications and TCD. The other one, Target Information Database (TID), stores detailed information about ships', aircrafts' and troops' characteristics. Both databases are represented in figure 3 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 database.

To take care of all its tasks, DOSM has two managers: the Domain Manager and the Event Manager (Figure 3).

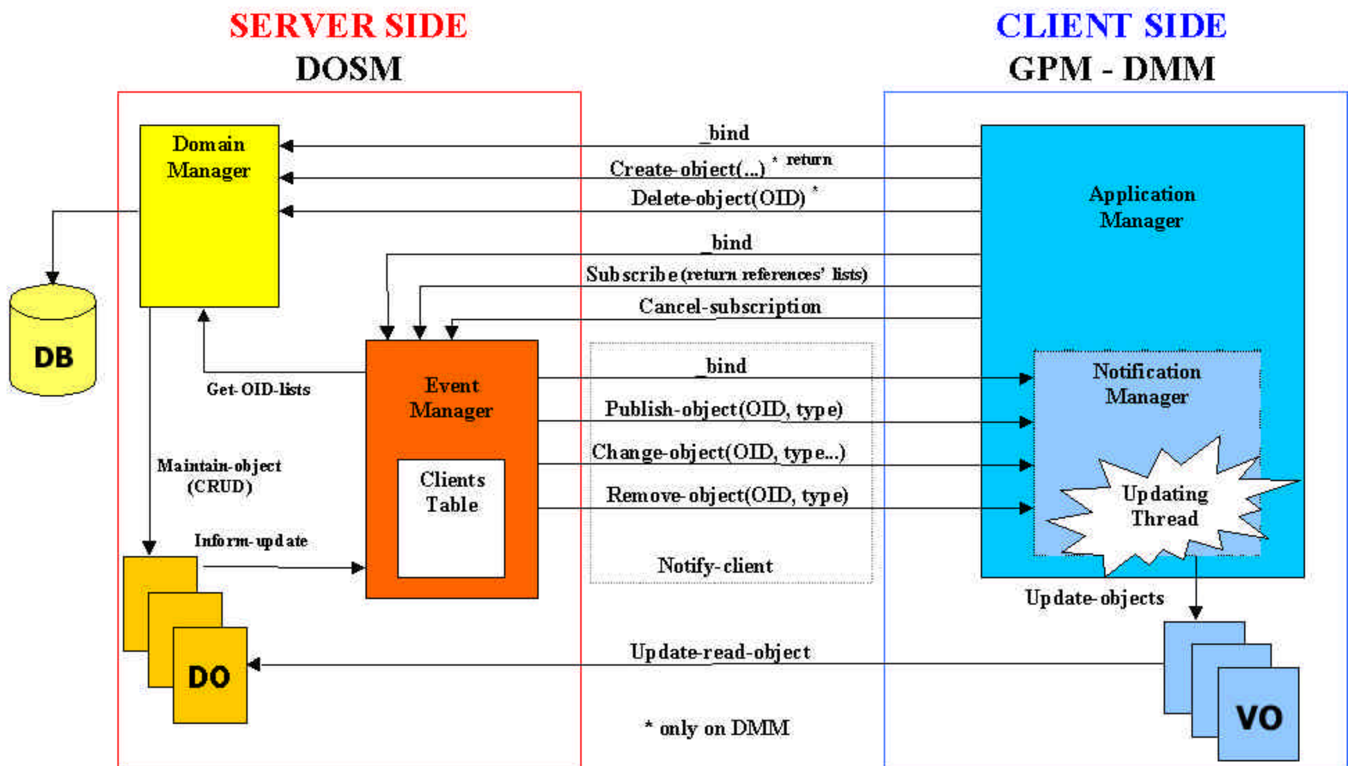

Figure 3 - The SATO Distributed Objects Implementation

The Domain Manager is mostly responsible for keeping the DO's in memory, talking to the databases and keeping them updated. It uses both CORBA and SQL for communicating with the clients and the databases, respectively.

The Event Manager takes care of the DOs' distribution. It provides the data necessary for initializing the clients, receives their information about changes on DO's, keeps them informed of every change made by other clients and keeps track of the connected clients and their connections' status.

The Application Manager

On the client side, the Application Manager does the essential work. Whenever a client is started, it establishes connection with the server, preparing to communicate, 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 interest for its client type. The Application Manager has a special interface, called Notification Manager (NoM), which receives, and processes, all information of changes from DOSM. Since the number of such notifications is very large, it has separate threads to conduct the changes on each VO. AppM has another significant function. On every notification sent by the server, it is responsible for the acknowledgement. If for any reason, the acknowledgement, after a certain time, is not received by DOSM, it deactivates the client. This means, for DOSM, that the client is no longer able to receive notifications and it is out of synchronization. When this happens, the next time this deactivated client tries to connect to the server, it will be forced by it to proceed with a full synch, i.e., a subscription.

The Event Manager

The Event Manager is implemented as a Singleton [GAM95]. Clients associated to it can receive 4 types of events:

- DOSM available objects;
- New object;
- Changed object; and
- Removed object.

Each client associated to DOSM has an event queue associated with it. Every new event that has to be sent to the client is placed in the queue and is associated with one specific domain object, called EventTarget. In the case of the InitializationEvent, the EventTarget is the Domain Manager, which keeps track of all DO's. The static structure of the EventManager and related objects is shown in figure 4.

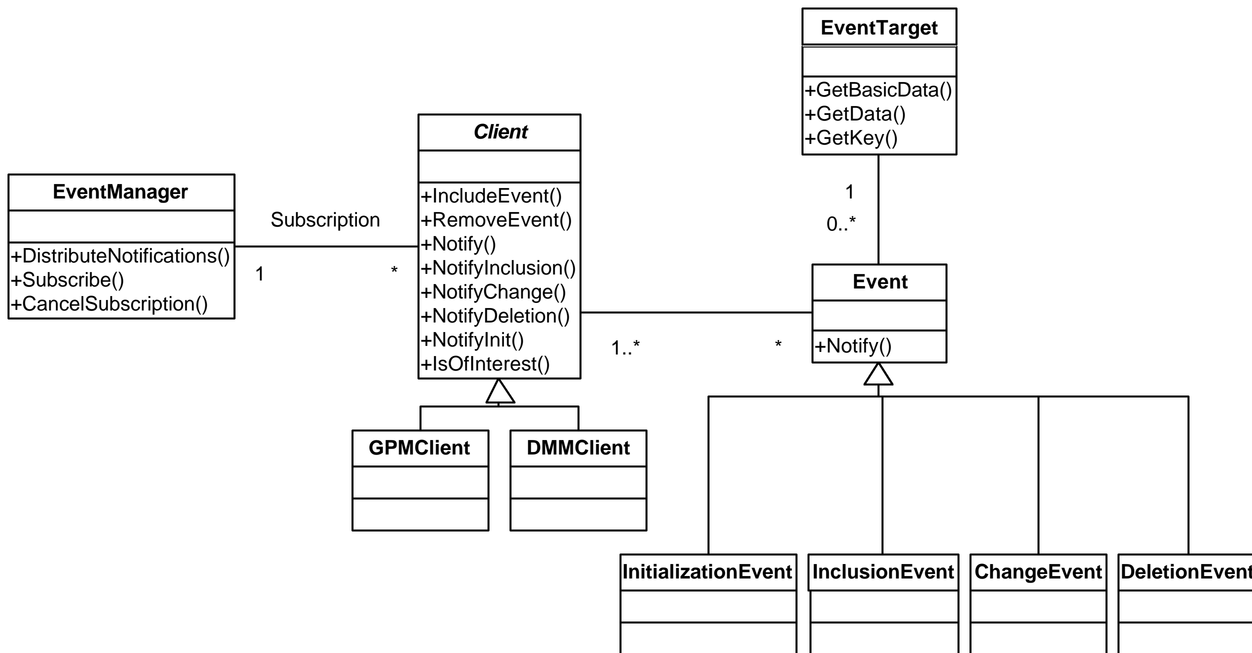

Figure 4 – Event Manager and related classes

The client's subscription process starts with a bind on the EventManager. Using the latter's proxy, the client subscribes the DOSM services. According to the client type (informed in the subscription), the EM creates an appropriate concrete Client. The notifications follow the Distributed Callback pattern [MOW97], avoiding the need of subsequent binds. When the subscription is finished, the client starts waiting for notifications. The DOSM creates an InitializationEvent then. This event is associated with the Domain Manager, placed in the client's event queue and DOSM starts its notification. Only after this notification is finished, the client is included in the list of clients able to accept regular notifications. A separate thread treats each client event and priority is adjusted to avoid starvation of a client. Figure 5 shows the Sequence Diagram for the Subscription process.

Although implemented this way now, the EM was not planned to work this way. On its first implementation, the subscription data was returned as out parameters of the subscription method, instead of as an event created by the DOSM (and sent when it is "free"). As originally implemented, it turned out to be ineffective. The preparation of the client's initialization data is a costly process, and letting the control of when to perform it on the client did not work. DOSM was eventually interrupted, during an important task, to answer a new client connection.

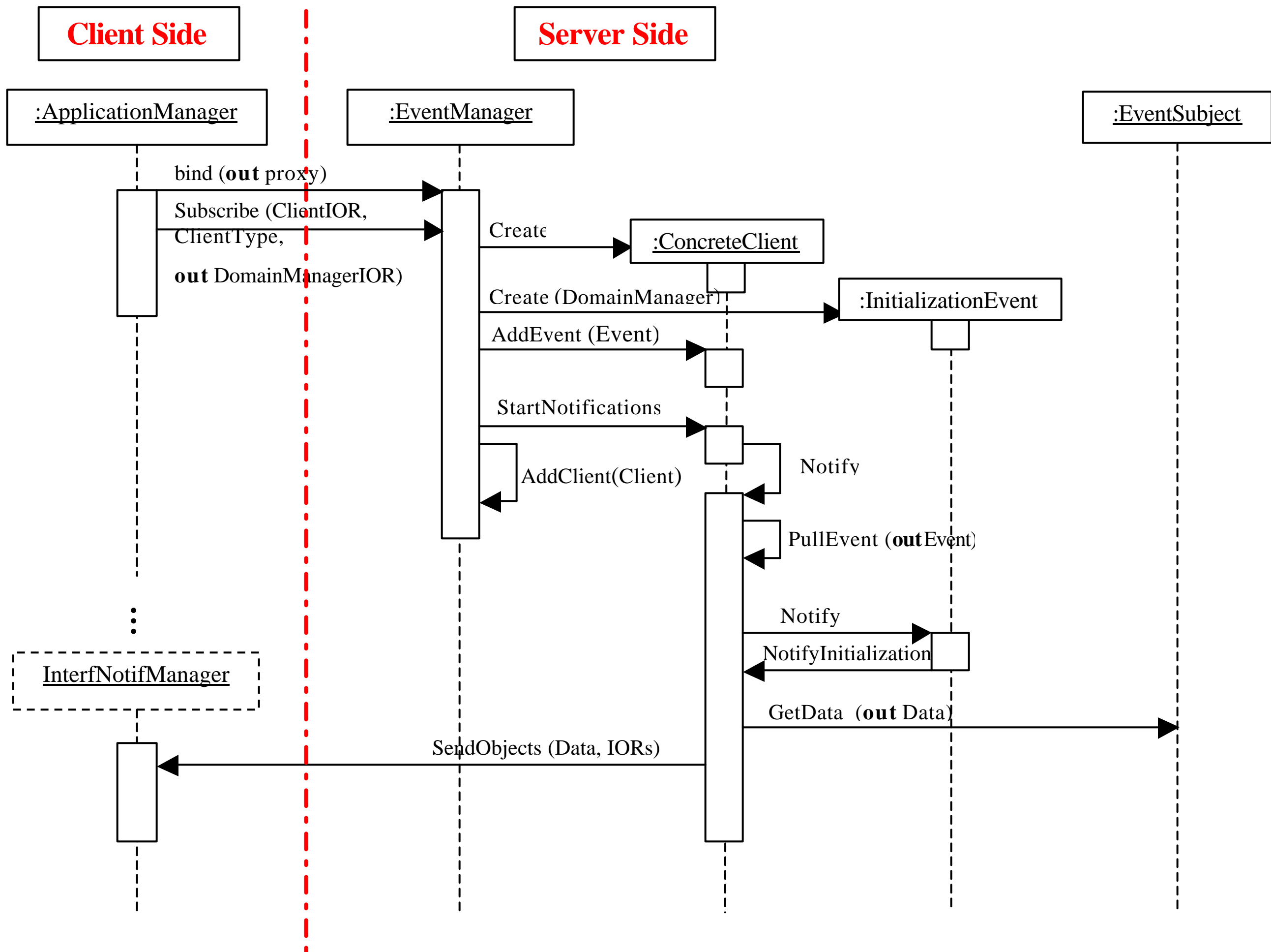

Figure 5 – Subscription Sequence Diagram

Fault Tolerance

System vulnerability resides on the DOSM (EM) since it is responsible for all the communications with the clients. Three basic problems arise:

- Dependency of 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 multi-threaded object, with fault and deadlock detection algorithms.

Typically, EM controls one notification thread per client. These threads can share the object the notification refers to, though. Semaphores are used to guarantee exclusive access to the shared object 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 6.

Every client starts, by subscription, on the *Connected* state. When *Connected*, its thread has the highest priority. At the first detected error on a notification to a client, it transitions to the *Faulty* state. On this state, since there is a suspicion that a communication problem is occurring, the thread has its priority reduced, to allow the normal operation of the other clients. However, the notification is not lost – EM will keep trying to deliver it, until a certain number of unsuccessful attempts. In addition, during this time, new events that may arrive are placed on the client's queue. When the limit of attempts is reached, the client transitions to the *Not Updated* state. If, before reaching the limit, a notification succeeds, the client transitions back to *Connected* and the events in the queue are all dispatched. On the *Not Updated*

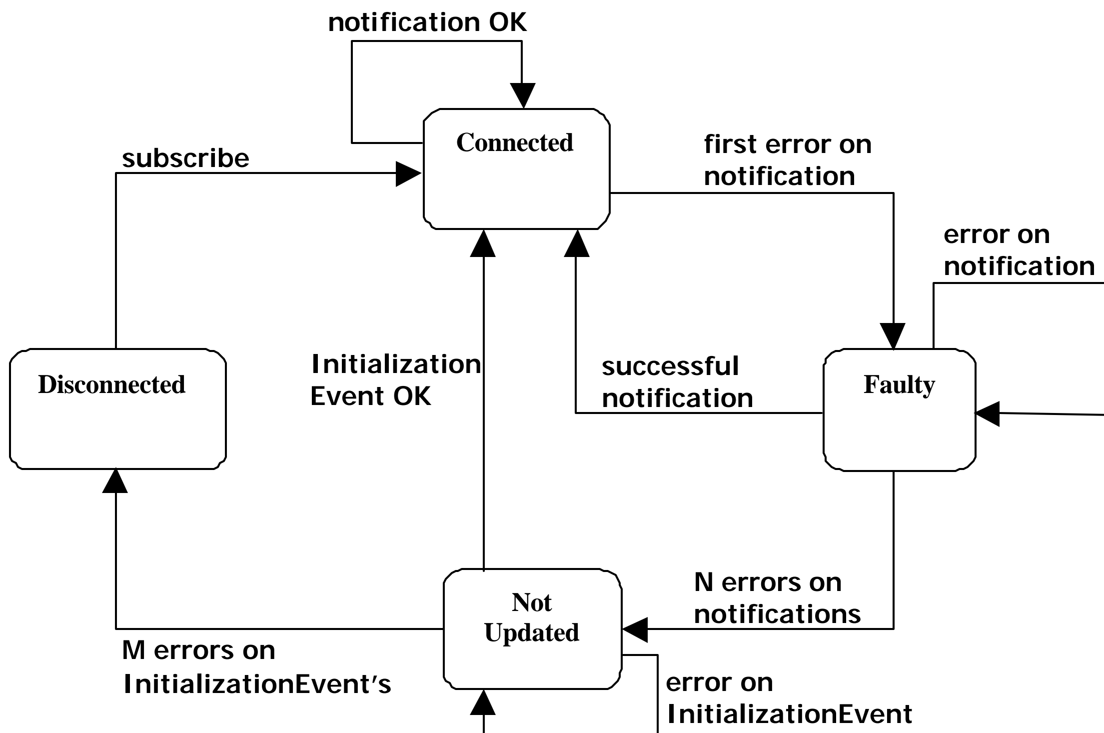

Figure 6 – EM Client's State Transition Diagram

state, its event queue is cleaned and EM tries to send an InitializationEvent (like on the Subscription process) but now, with a lower priority, since the chance of having lost the connection is higher. Again,

if a successful attempt is obtained, the client returns to *Connected*. Else, if an established limit of attempts is reached, the client transitions to the *Disconnected* state. Then, only a new subscription brings the client back to *Connected*.

The limits (M and N) shown on figure 6, were determined on system's trials and can be adjusted to specific network configurations, as this proved necessary.

5 SUMMARY AND FUTURE WORK

This paper presented the Mercury Project and the architecture created for its main subsystem – SATO. The solution adopted was a mixing of several architectural styles. Although, as mentioned before, the performance of the system was not benchmarked, it has been quite acceptable to the users. Initially, it was not what the users expected but, after tuning up the subscription process, all fell into place. The Event service [BAK97] implementations tested (ORBIX and COOL Chorus), at the time, were not satisfactory. Some problems, especially with the multithreading versions were faced and so, the event management was fully implemented for the project. This choice was also made because we believed that a more flexible filtering mechanism would be required [CUG98] and special adjustments would have to be made for the final version of the system.

As shown above, the Mercury Project faces several demanding requirements. Implementing the EM and adjusting its behavior to the networks it runs on, was a tough job. The subscription process is onerous both on clients and on DOSM. At this stage, performance was a problem and had to be achieved at any cost. 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, which promises to be a challenging task. 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. This requires the use of optimization algorithms, as minimum cost routing, and tagged events, which will allow the client to update itself when changing from one DOSM to another.

ACKNOWLEDGEMENTS

The expertise of Jose Gomes from the Brazilian Navy Research Lab (IPqM), gained in the construction of the Brazilian Navy Academy Navigation Simulator was of great value.

REFERENCES

1. [BAK97] Baker, S. CORBA Distributed Objects. *ACM Press, 1997.*
2. [BRO97] Brown, K. Crossing Chasms – The Architectural Patterns. www.ksccary.com
3. [CAR99] Carzaniga, A. Rosenblum, D.S. Wolf, A.L. Challenges for Distributed Event Services: Scalability vs. Expressiveness. *Proceedings of EDO 99, IEEE Press.*
4. [CUG98] Cugola, G. Di Nitto, E. Fuggeta, A. Exploiting an Event-Based Architecture to Develop Complex Distributed Systems. *Proceedings of ICSE 98, IEEE Press.*
5. [EMM99] Roodyn, N. Emmerich, W. An Architectural Style for Multiple Real-Time Data Feeds. 1999.
6. [GAM95] Gamma, E. Helm, R. Ralph, J. Vlissides, J. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Co., 1995.
7. [IONA] IONA Technologies Ltd. ORBIX Programming Guide. 1998

8. [KRA88] Krasner, G.E. Pope, S.T. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *ParcPlace Systems, 1988.*
9. [MOW97] Mowbray, T.J. Malveau, R.C. CORBA Design Patterns. *John Wiley & Sons, 1997.*
10. [PRESS] Pressman, R.S. Software Engineering – A Practitioner's Approach. 4th edition. *McGraw-Hill, 1997.*
11. [ROD01] Rodrigues, J.A., Cruz, S.O. et al A Command and Control Support System using CORBA. *Proceedings of ICDCS 2001, IEEE Press.*
12. [ROO99] Roodyn, N. A Software Architecture for A Real Time Data Distributed Objects System. *Proceedings of EDO 99, IEEE Press.*
13. [SCH99] Shönhage, B. Eliëns, A. From Distributed Objects to Architectural Styles. *Proceedings of EDO 99, IEEE Press.*
14. [SHAWM] Shaw, M. Garlan, D. Software Architecture – Perspectives on an Emerging Discipline. *Prentice Hall, 1996.*
15. [URB99] Urban, S.D. Fu, L. Shah, J.J. Harter, E. Bluhm, T. Hartman, B. The Implementation and Evaluation of the Use of CORBA in an Engineering Design Application. *Proceedings of EDO 99, IEEE Press.*