Publisher Framework (PFW)

Judith Klein
judith.klein@lmco.com
Lockheed Martin
9211 Corporate Boulevard, Rockville, MD, 20850, USA

Drasko Sotirovski
drasko@acm.org
Raytheon Canada Ltd
#150-13575 Commerce Parkway, Richmond, BC, V6V 2L1, CANADA

Abstract

One of the lasting challenges in building distributed fault tolerant systems is keeping application code size and complexity down. This can be done by capturing the nuances of distributed computing environment and redundant fault tolerant elements into a common infrastructure layer, thus factoring the code that would otherwise need to be written again and again by each distributed fault tolerant software component. When the application code has many complexities, and Air Traffic Control (ATC) is certainly one such example, achieving this goal becomes paramount.

Under a project called En Route Automation Modernization (ERAM), the Federal Aviation Administration (FAA) is developing a replacement for its aging en route assets. At the same time, a foundation is being created for the anticipated future enhancements, driven by the projected increase in air traffic. At the core of the ERAM design is a distributed object oriented (OO) framework called Publisher FrameWork (PFW), which is ERAM's answer to the aforementioned OO challenge. This paper describes the PFW properties, the experiences with it accumulated through the first build of the ERAM program, and its applicability to fault tolerant computing.

Introduction

Distributed computing is 20+ years old, but it only took the main stage with the explosion of networking and the internet in particular. The outburst of distributed computing frameworks (CORBA, J2EE, .NET, to name just the few most popular) is no surprise and one should expect a plethora of distributed computing environments before some of the difficult issues in distributing computing are settled satisfactorily. From this point of view, PFW is no exception: just one of the many! Although true to an extent, PFW is also more: an attempt to raise the bar and attack not only the relatively simple issues of messaging/dispatching, but also some rather difficult issues related to encapsulation, extensibility, scalability, performance, and availability, all with no significant increase in application code size and complexity. PFW is a lightweight framework: it enables applications to focus solely on the application domain.

In the subsequent sections, we describe the software component methodology we have followed and the resulting need for publication services, mirror storage and subscriber synchronization. To keep our experience report focused, we are not going to compare PFW with CORBA or J2EE, which have since started including fault tolerant elements. We hope to provide enough insight into PFW for the readers to make that comparison on their own. The fact that PFW provides

cross-language support (ERAM components are split between Ada and C++) was also ruled outside the scope of this particular discussion.

Distributed Software Components

Our expertise is in building large scale, distributed, fault tolerant, near real-time systems. The application domain we've been concentrating on for over the last 15+ years has been air traffic control. We, at Lockheed Martin, had already developed and fielded a robust infrastructure called FlightDeck™[1] (see **Figure 1 Side-Bar on FlightDeck Middleware**), which provides a rich set of relevant services. We proceeded to concentrate on the domain of air traffic control. Here we started with a data model and ended with a set of software components, a description of how the components should behave, with a common behavioral pattern and a strict dependency hierarchy for ease in building the system. This inevitably[2] led to PFW, an infrastructure extension which captures and reinforces this pattern.

Supported platforms: IBM AIX™, Sun Solaris™
Supported Programs: China CAA's Air Traffic Control Automation System (ATCAS); FAA's Display System Replacement (DSR); UK NATS' New En Route Center (NERC); US FAA's User Request and Evaluation Tool (URET); US FAA's En Route Automation Modernization (ERAM).

Communication Services, featuring:

- broadcast, multicast, and connected data transfer with support for redundant servers
- automatic division of messages to network frame size
- naming services which isolate applications from the system topology
- network overload protection
- higher level abstractions are also supported:
  - publish/subscribe
  - client/server
- reduced overhead with minimal data movement

Availability services, featuring:

- detection of software crashes (<1 ms) and hangs (adaptable by application)
- detection of processor failures (<1 sec, adaptable)
- recovery actions (adaptable) which can be initiated either locally, within server groups, or across the system (<1 sec, adaptable)
- hardware certification at IPL and at adapted intervals

System time services, featuring:

- time of day clocks
- multiple simulated clocks
- highly synchronized system clock (20 ms or better, adaptable)

System recording/analysis, featuring:

- online and offline analysis tools
- adaptable, command-able, and event driven data collection

System management services, featuring:

- centralized monitor and control capability with multiple command modes and verification methods, and hierarchical status
- low impact distribution using paced broadcast to allow continuous mission function
- checksuming during distribution and processor IPL
- management of multiple releases

Application builders, featuring:

- event services, which provides an integrated interface to the system communication and time services
- data checkpointing services
- synchronized distribution of shared memory to selected nodes
- structured support for command and control of state and intent of user-defined system elements, each of which may be independent or part of a complex hierarchy of elements related via a set of Boolean operators
- standby application support for redundant applications

Figure 1 Side-Bar on FlightDeck Middleware

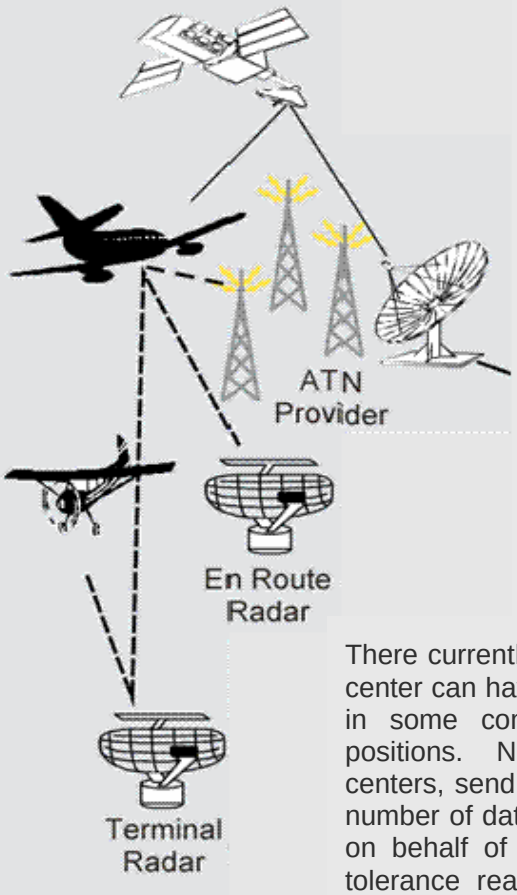
[1] FlightDeck™ is a trademark of Lockheed Martin.
[2] One of us, while working on the Canadian Automated Air Traffic System (CAATS), developed a common middleware layer with similar properties to PFW (see references [1] and [2]), leading us to believe that these system types are conducive to such software designs.

Component Definition

A *component* is a logical grouping of software whose definition is based on the problem domain. A software component provides a cohesive set of services, exports a well defined interface (application programming interface, API); it encapsulates implementation details (internal databases, data structures and internal functions are hidden from the client). Furthermore, software components can be independently developed and tested.

Methodology Used To Define Components

Objects in the problem domain (e.g., airport, route, target report) form natural dependencies and relationships (e.g., a flight plan has a departure point, a destination and a route). Following this well established software engineering principle, we grouped cohesive objects into software components in a manner in which interfaces between components were minimized. In the process, we established dependency rules among components: a hierarchy of components which is strictly enforced in the build process. Component definitions are advertised in terms of provided services on the encapsulated data objects. Added benefits of this approach are: extensibility (e.g., additional components are built using existing components, promoting re-use of existing components) and ease of component replacement (as long as the API is invariant, the implementation can change).



FAA En Route Air Traffic Control

The current Host Computer System operating in the Federal Aviation Administration (FAA)'s En Route centers has been the backbone of the US National Airspace System (NAS) for thirty five years. In 2003 the FAA awarded a contract to Lockheed Martin to replace the existing NAS Host and DARC (Direct Access Radar Channel).

ERAM adds improved capabilities of NAS Architecture, Free Flight Initiatives, Advanced Communication, Navigation, Surveillance, Information Management, and Decision Support Technologies that can now be applied to Air Traffic Management. It will replace the existing mainframe-centric Host architecture with a state of the art open and supportable environment.

There currently are twenty en route centers in the Continental U.S. A center can have hundreds of air traffic controllers working side by side in some combination of Radar, Data, Assistant and Specialist positions. Numerous external interfaces, including other en route centers, send a constant stream of inputs and expect similar data. A number of data servers perform the bulk of the algorithmic processing on behalf of one en route center; they are all redundant for fault tolerance reasons. Workstations are dedicated to the controllers. Throughput and response time requirements place challenging constraints on the system architecture and design.

Component Deployment to Physical Nodes

An *executable* is a physical grouping of software, an independently start-able, stoppable program: with Unix, this is a single process that may includes multiple threads. Definition of executables is based on the knowledge of the physical architecture and with fault tolerance in mind: the executable is a unit of failure, while a thread is a unit of concurrency. Executables we build in the air traffic control domain, follow the event-driven model:

a) Multiple threads of execution are packaged into the same executable.
b) Each thread of execution is in a forever loop waiting for events, servicing each event in priority order, making synchronous (e.g., library calls) and asynchronous service requests (e.g., to a service residing on a different node).
c) For asynchronous requests the address of a callback procedure is provided to the service: when the service completes, the callback is invoked with the results.

A component spans executables. Parts of a component can be bound into a server/publisher executable resident on one node, while other parts of the same component are bound into a client/subscriber executable resident on a different node (this is similar to the J2EE concepts of local and remote interfaces or CORBA proxies and skeletons). The data exchange between the publisher of the component and the subscriber is internal to the component: the format of that exchange can be modified without impacting the users of the component's services (since they access the component strictly by using the API); in other words, whether the data is encoded for transmission (a.k.a. serialized) into XML format, binary format, or something else, the users of the components are unaffected as long as the APIs used to access the objects (attributes, methods) are constant. Publishers of multiple components can be bound together into a single executable, along with a number of other components' client-side code to achieve the application mission (e.g., detecting conflicts between aircraft).

We describe the physical software architecture by showing the placement of defined executables on nodes (processors) of the hardware architecture and by showing the parts of components that were bound together to form each executable.

Lockheed Martin defined and implemented a first version of the component model (i.e. the collection of components needed to implement an air traffic control system, along with their interfaces and interactions) for use in "User Request Evaluation Tool". This tool is now deployed nation-wide; it automatically predicts and notifies controllers of conflicts between aircraft or special activity airspace. The system also allows controllers to quickly determine whether proposed flight path changes will conflict with en route traffic or airspace. By allowing controllers to evaluate route change requests and to assign conflict free routing, the airspace users are able to save both time and fuel.

For the first implementation each component's behavior was described from an architecture standpoint – the pattern was defined, but no common framework was provided. The resulting implementation proved that the concepts were solid. However, we noticed that there was large variation in the implementation specifics of components, as well as some degree of code duplication for common component behavior. We concluded that benefits could be reaped from factoring out the common behavior into a common framework:

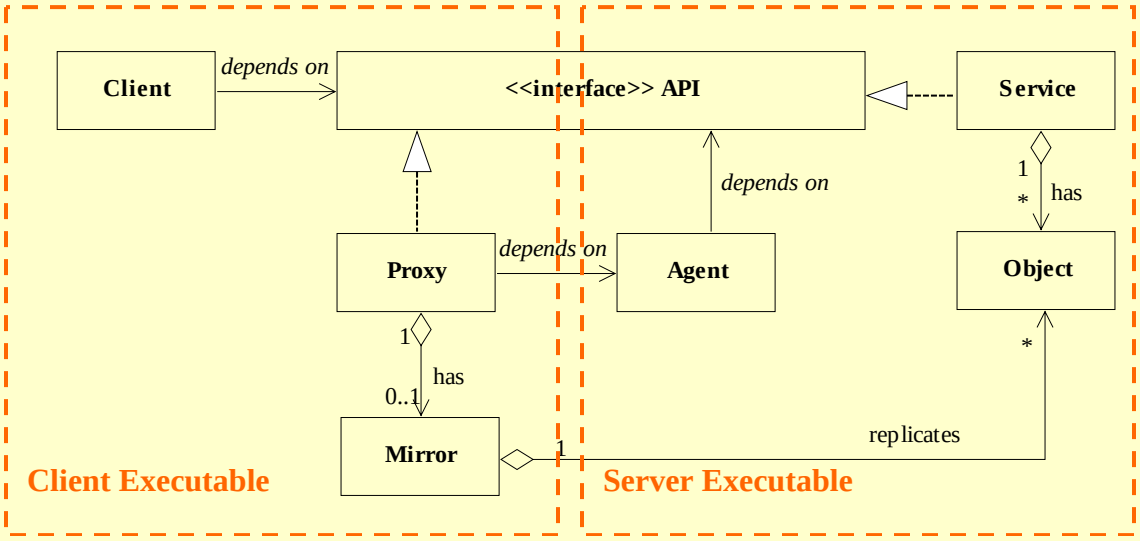
a) Overall code size could be reduced.

b) Errors in implementing the basics of the component framework could be eliminated when correcting them once in the common framework.
c) Components would be more maintainable since they would be concentrating on the domain expertise rather than on framework matters.

Such observations led us to the development of the Publisher FrameWork (PFW) on which most[3] components in the air traffic control domain of ERAM are now built. As we prototyped PFW, we found more and more common behavior to be factored and included into PFW, such as data redundancy for fault tolerance.

Publisher FrameWork (PFW)

The Publisher FrameWork (PFW) provides a framework for uniform, consistent development of software components. The design pattern (see **Figure 2 Anatomy of a PFW Distributed Service**) implements support for:



PFW provides a Service (resident on a server) with a distributed front-end facade: Mirror provides synchronous read access; Proxy asynchronous update access to server's internal database. The data published by the server is provided to the client for processing and can be stored in the client if needed (instead of being replicated in the Mirror).

Figure 2 Anatomy of a PFW Distributed Service

a) A *server* to publish objects to subscribers and to process requests from clients.
b) An *agent* acting as a local subscriber to receive published objects, translate them into messages and multicast them to all remote subscribers. The use of multicast mechanism makes PFW scalable to the hundreds of positions that must be supported in an en-route center.
c) A *proxy* to receive multicast messages, translate them back into objects and republish them to local clients. The component user, when notified that an update has arrived, is guaranteed

[3] Some legacy components were not converted to using PFW to minimize change of working components.

that the mirror is current, i.e., the update has been applied to the content of the mirror; therefore the component user can make queries against the object attributes from within the context of the callback. Additionally, the proxy facilitates requests from the clients to the server; a watchdog timer is used to monitor the arrival of a timely reply from the Server – the client is therefore guaranteed to be notified either about the completion of the request or about its timeout.

d) A *mirror* to augment the proxy by retaining a copy of the data published by the server for use in local queries. The existence of the mirror provides the client with the convenience of accessing the data not only when the information is received, but also as part of other processing, such as the expiration of a timer. In response to a request to update a server object, the mirror is updated before the confirmation is delivered to requestor, so that the requestor can reach into the mirror and access object attributes and methods with the assurance that the object is up-to-date.

In ERAM, for which PFW was developed, all the software components are known – their names (server name, published data-stream name) and APIs are documented; all necessary discovery is done by infrastructure alone, which locates the registered server(s), selects the Primary[4] executable of that server, and delivers client requests to the server.

Note that there is a plethora of requirements implemented by PFW on behalf of all components that are less interesting to describe in this paper, yet helpful to the component implementers just the same; one example is invoking the recording service and error reporting service (to log commonly recorded events and data) for detected errors.

Mirror and Original, Queries and Updates

As introduced above, a mirror encapsulates client-side storage of object replica matching the original objects of the server's internal database. In other words, the client-side proxy subscribes to (registers interest in) the data-stream published by the server; when a publication is received, the proxy first updates the object replica in the mirror storage; other local clients can then safely be invoked with the guarantee that the mirror is current and consistent with the server's internal database. Having a mirror presents the advantage of being able to perform synchronous queries against the local object cache.

Requests for update of an object are asynchronous: the request is forwarded to the server (whether local to the executable or remote); a callback procedure is provided so that it can be invoked when the results of the asynchronous update are received. All updates to an object are performed only by the owner of the object (the server/publisher, not a mirror – merely a copy of the object is stored in the mirror). A simple and robust approach insures consistency of the data throughout.

Finally, local clients can be notified of changes (in addition to being able to query the local mirror storage). There are two kinds of registration:

[4] See details on Primary vs. Standby in **Figure 1 Side-Bar on FlightDeck Middleware**.

(a) For all objects of a class, resulting in the registrant being notified whenever an object is created (added to the mirror storage), deleted (removed from the mirror storage) or updated (modified in the mirror storage);
(b) For a specific instance, resulting in the registrant being notified whenever that instance is modified, including deletion.

In conclusion, the object replica PFW maintains in the mirror storage is, from a client's perspective, virtually indistinguishable from the original: it can be observed through registration/notification, queried and updated – albeit in an asynchronous fashion. This provides near-perfect object location transparency. Only the fact that update methods are asynchronous hint to a client that the target object may or may not be local. In any other respect, the client-side replica is indistinguishable from the original object.

Fault Tolerance

A primary-standby pattern is a standard arrangement for high integrity systems. The goal is always one and the same: to provide full protection from hardware faults and protection from at least transient software faults. In industrial applications, like ATC systems, the failure model is always fail-safe, but its dynamic characteristics vary from component to component, depending on the component criticality and the required switchover time. Components in ERAM range from active fail-silent (all redundant elements are active and at all times ready to provide services) to passive fail-stop (redundant elements are available but become active only after the primary had failed). In other words, primary provides the service with a (more-or-less hot) standby ready to replace it in case of a failure. In addition, if a service fails when no standby is available, the service can be restored with little or no loss in functionality from the checkpoint data that the primary and standby have saved on disk.

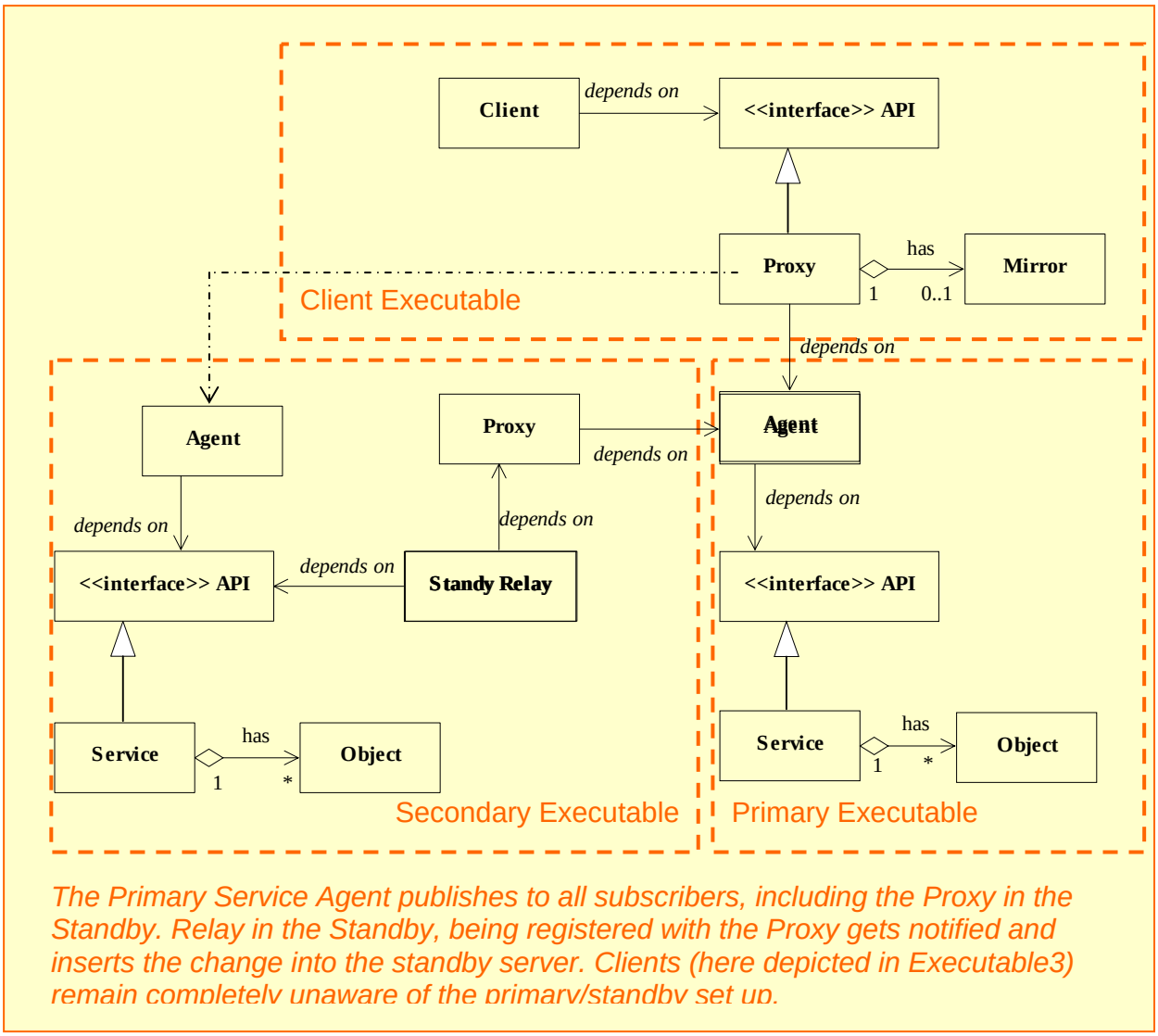


Figure 3 PFW Support for Fault Tolerance

When prototyping PFW, we concluded that we can include support for the required spectrum of Standby designs by simply making the Standby a Subscriber to the very data-stream its Primary is publishing, as described in **Figure 3 PFW Support for Fault Tolerance**. The essence of PFW fault tolerant behavior, from a client perspective, is best described by simply stating: the client side object replicas are transparently rewired from the originals in the failed primary to the new originals in the new primary. This is however, as many a reader will know from experience, easier said than done. PFW implements near perfect transparency relative to both outstanding requests and clients registered for notification. In the following paragraphs, we describe some of the usual yet intricate obstacles in achieving these goals (and give the meaning of 'near-perfect transparency', 'more-or less hot standby', etc.).

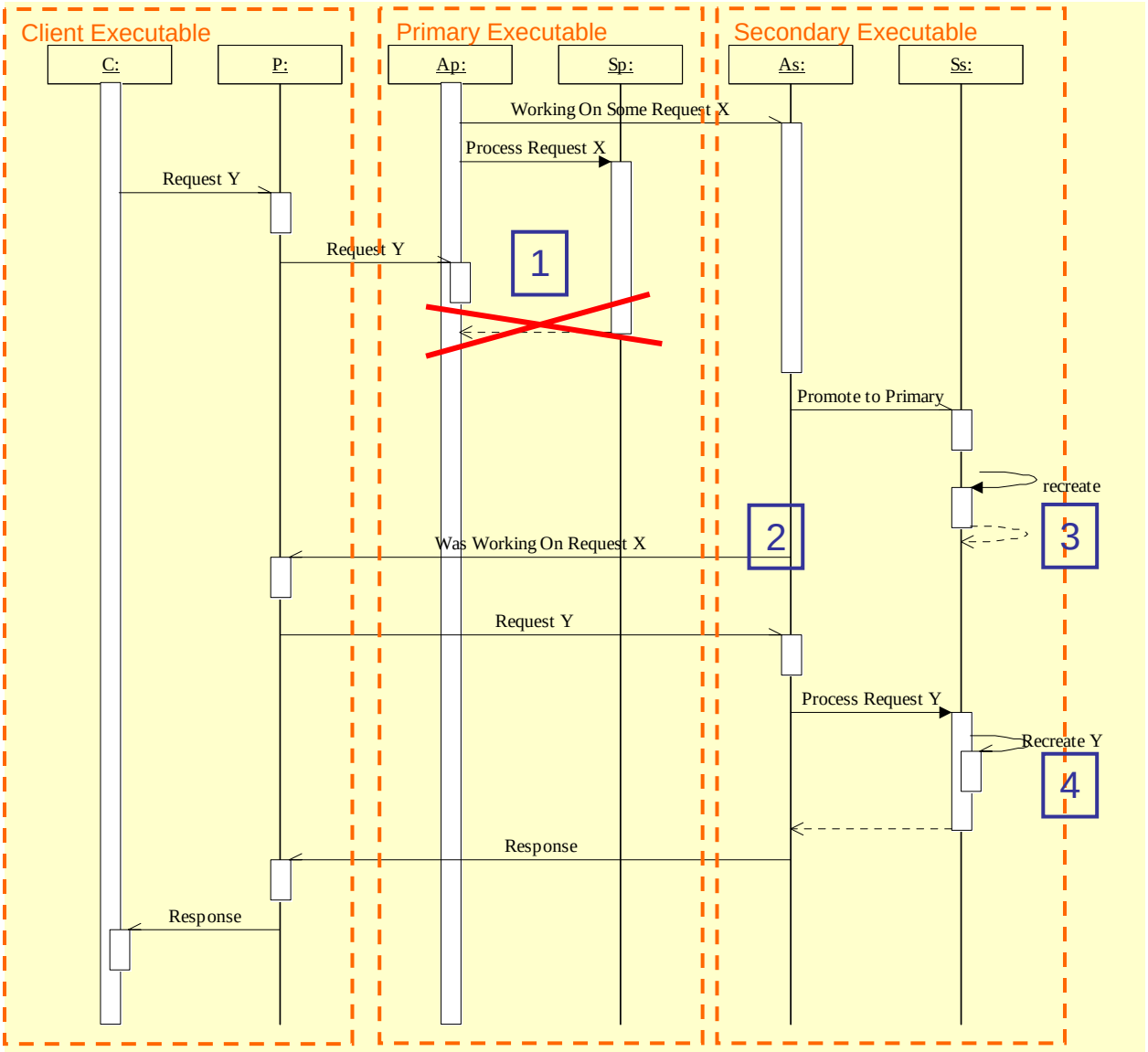
Not only redundant servers fail, but clients can fail too. Having no redundancy, such computing elements need to get back in sync with the rest of the distributed environment, which poses additional challenges; in particular for getting the new members back in sync with no adverse

performance impact on the system. PFW approach to solving this issue is deemed outside the scope of this particular discussion and readers interested in it are referred to [3].

Enhancements under Consideration

At the time the primary service failed, it may have been processing a request from a client, with more requests stuck in its input queue. To make the switchover transparent to clients, all these requests need to be processed, with the new primary taking over where the old primary stopped. Obviously, the request being processed at the time the Primary failed may be the killer request and automatically reprocessing it exposes the system to the common mode failure. All other requests are safe to reprocess (in particular if addressed to other object instances). PFW currently makes clients with outstanding requests decide. Clients will receive a time-out on the request and must decide to resubmit it or not. PFW improvements under consideration include a means for clients to find out if the request is suspected to be the killer (i.e. if this very request was processed when the primary failed) as well as automatic resubmission of the requests stuck in the input queue of the primary at the time it died, provided they are not directed to the same object as the killer request. See points 1, 2 in **Figure 4 PFW Facilitated Switchover of a Component**.

Objects in the server are not only state; in addition to the state, an object may have some *dynamic context*, e.g. an outstanding timer to do something. This dynamic context can of course be mirrored by the *hot* standby. However, this makes primary and standby go through the same computational history and increases the likelihood of simultaneous primary-standby failure. For this reason, many system designs (including PFW) opt for recreating this dynamic context upon switchover. So far so good, but processing requests is only possible *after* the dynamic context is re-established. Yet reestablishing the dynamic context for thousands of objects can take a significant amount of time resulting in an unacceptable hiccup in system performance immediately after a switchover. PFW's answer to this significant challenge is described in **Figure 4 PFW Facilitated Switchover of a Component** (see points 3, 4).



(1) At the time primary server fails, it may be processing a request from one of the clients with other requests waiting in its inbound queue.
(2) Secondary server, after FlightDeck had rewired the communication, publishes the killer request so that all clients with outstanding requests can resubmit them – except for the client holding the killer request.
(3) Secondary is using the system idle time to recreate the dynamic context of all objects, a quantum at a time.
(4) A request received for an object that has not been restored yet triggers the recreation and only then proceeds to executing the request.

Figure 4 PFW Facilitated Switchover of a Component

Conclusions

High availability systems have needs above and beyond the functionality provided by the contemporary commercial middleware (CORBA, J2EE, .NET, to name a few). ERAM successfully serves these needs through PFW by extending the middleware services to provide location transparency and fault tolerance. As additional common behavioral patterns become apparent, the authors will consider incorporating capabilities to address these into future PFW releases. In return, an important part of common and intricate implementation is factored while leaving the application domain to remain focused on solving domain issues.

Acknowledgements

The authors acknowledge the following people who made significant contributions to the concept definition, prototype, design and development of PFW:

- **Tim Donovan**, Software Architect with Raytheon Integrated Defense Systems, Tewksbury, MA
- **Sam Carnicelli,** Chief Designer with Lockheed Martin Transportation and Security Solutions, Cato, NY
- This work was performed under the contract from the Federal Aviation Administration, DTFA01-03-C-00015. The support and review from **Jeff O'Leary**, FAA ERAM Product Team Software Lead is especially noted and appreciated.

References

[1] Thompson, C., J., Celier, V., *DVM: an object-oriented framework for building large distributed Ada systems*, TRI-Ada '95, Anaheim, CA

[2] Sotirovski, D., *Towards Fault-tolerant Software Architectures,* Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands.

[3] Sotirovski, D., *Time Horizon in Distributed Object Societies,* a companion paper submitted at SIGADA 2006.

About the Authors

Judith Klein is a certified systems architect at Lockheed Martin Transportation and Security Solutions. She has 28 years of experience developing distributed real-time systems of various sizes in different domains; the last 15 years have been focused on air traffic control. She has a BS in applied mathematics and computer science from Carnegie Mellon University in Pittsburgh, PA and an MS in technical management from Johns Hopkins University in Baltimore, MD. She is a senior member of the IEEE and a member of the Association for Computing Machinery.

Drasko Sotirovski is a software architect at Raytheon Systems Canada. He has 27 years of experience in developing large-scale real-time software for defense, simulation, transport, and telecommunication systems for several European and North American customers. His research interests are software architecture and distributed object-oriented technologies. He received a BSc in technical physics and computer science from Elektrotehnicki Fakultet u Beogradu, Yugoslavia. He is a member of the IEEE Computer Society and the Association for Computing Machinery. He is also a PEng with the new CSED (Computer and Software Engineering Division) branch of APEG BC.