Macrothink Institute™

Network Protocols and Algorithms

ISSN 1943-3581

2010, Vol. 2, No. 3

Data distribution technologies in wide area systems: lessons learned from SWIM-SUIT project

G. Carrozza, D.Di Crescenzo, A. Napolitano, A.Strano

SESM S.c.a.r.l.

Via Circumvallazione Esterna di Napoli 80014

Tel:+39 081 8180356 E-mail: {*gcarozza; ddicrescenzo; anapolitano; astrano*}@sesm.it

Received: September 27, 2010 Accepted: October 5, 2010 DOI: 10.5296/npa.v2i3.473

Abstract

To face the increasing air traffic demand, intended to dramatically grow in the next years, Europe and US are going to develop new and modernized Air Traffic Management (ATM) systems, based on novel and integrated concepts and technologies. Hence, global interoperability that is an essential goal when planning the development of ATM air/ground applications and systems, becomes paramount. To date, the management of different types of air traffic information, referring to service and subsystem specific requirements, makes the current systems not suitable and insufficient for integrating and sharing of relevant information. SWIM is the new information infrastructure, which will connect all ATM stakeholders, to allow seamless information interchange for improved decision-making. Stemming from the experience gained from SELEX-SI[1] during the development and test phases of a pilot European research initiative, this paper describes the architecture of a prototype implemented to investigate the technological feasibility and applicability of the SWIM principles. It provides a practical technical report aiming to highlight the achieved results and to share the great know how coming from this experience with researchers and practitioners in this field.

Keywords: Air Traffic Management, Critical systems, Data Distribution Service, Large scale systems.

[1]SELEX-Sistemi Integrati, FINMECCANICA group, is one of the European leaders in field of ATM. http://www.selex-si.com

1. Introduction

Air Traffic Management (ATM) domain is currently moving towards global interoperability and longing for a Single European Sky [1], aiming at increasing the overall ATM efficiency by setting up common rules and standards. This will let at European airspace to be no longer constrained within national borders [2]. To achieve this ambitious goal, several stakeholders, i.e.,, airports, airlines, military air defense, Area Control Centers (ACC) and Air Navigation Service Providers (ANSP), must be allowed to easily share information on a really large scale.

SWIM (System Wide Information Management) [3] is the world recognized initiative (both in Europe and USA respectively within the context of SESAR [4] and FAA [5] programmers) aiming to realize a middleware capable of enabling this seamless information sharing. It is meant to be the software infrastructure able to provide the one-for-all information model for data exchange and interoperability, as well as common interfaces to access specific services, between both similar administrative organizations, such as European Organization for the Safety of Air Navigation (EUROCONTROL) and different organizations and administrative domains, such as Federal Aviation Administration (FAA) and the same EUROCONTROL. To this aim, it is going to define a common dictionary in terms of data and services as well (e.g., flight and surveillance data domains use the results of the ICOG2 [6] project as baseline, and ASTERIX CAT 62 standard for surveillance information respectively [7]).

Moreover, it has in mind to use Commercial Off-The-Shelf (COTS) hardware and software to support a Service-Oriented Architecture (SOA) aiming to facilitate systems dynamic composition and to increase common situational awareness. Indeed, COTS's, or more generally OTS, are hardware or software components ready-made on the market, which on the one hand allow to reduce costs and time to market into the development of system of systems, but on the other hand give rise to manifold problems that have to be faced.

This COTS integration stands for an ambitious mission and poses several research and technological challenges to be take into account during the development of such a complex infrastructure. First, the envisioned system is a clear example of large scale software systems to be used in Large scale Complex Critical Infrastructures (LCCI) [8], hence great attention has to be paid to non functional requirements, like reliability, availability and security, as well as to safety and maintainability. Second, in the ATM scenario, systems to get interconnected are likely to be even spread across different countries and to be made up of interacting subsystems coming from several vendors. On the one hand, this imposes the definition of widely accepted standards. On the other, since these systems are expected to be alive for decades and updated asynchronously, it obliges toward a "design for change" approach [9] thanks to which changes into technologies and programming techniques do not impact on the software business logic and application. Third, existing systems of all the involved stakeholders, actually, represent very different baselines in terms of technology, capacity, deployment and size. Hence, integration challenges also rise with respect to these legacy solutions.

Thanks to their flexibility, transparency and location independence, Service Oriented Architecture (SOA) has been widely used so far to address similar challenges. However, the tricky task of data distribution and information sharing among remote instances of LCCI distributed systems requires loose coupling, as well as the ability of setting up specific requirements of Quality-of-Service (QoS). To do that, technologies supporting the publish/subscribe interaction pattern, like Java Messaging Service (JMS) [10] and Data Distribution Service (DDS) [11], are gaining more and more credit in the last recent years [12]. Both allow realizing a networking middleware for distributing heterogeneous data in real-time among several nodes and according with specific services requirements. Nodes, which produce information (publishers), create "topics" (e.g., temperature, location, pressure) and successively publish them. The middleware takes care of delivering the sample to all subscribers that declare an interest in that topic; asynchronous communication can be realized. This feature makes these middleware's attractive for ATM applications, in which the loose coupling constrain is a must. In this context, the aforementioned implementations of this middleware are actually the best representation of two philosophy, that is, commercial (DDS) and open-source (JMS) implementations that in this paper would be compared when they are exploited into LCCI systems.

Stemming from the above considerations and industrial experience gained by SELEX-SI in the context of the SWIM-SUIT FP6 European project [13], this paper focused the attention on illustrating the pilot initiative, which aims to investigate the technological solutions that could enable the development of SWIM through the realization of a SWIM prototype, namely SWIM-BOX. The main objective of this work is to give feedback on the usage of DDS and JMS to implement data distribution systems, and on providing hints to the providers on how to aid developers and systems integrators working on the these systems.

This experience raised plenty of insights both on methodological validation and verification approach, as well as on its design and implementation by means of different technologies.

2. Background

2.1 Data Distribution Service (DDS)

Data-Distribution Service (DDS) [11][14] is a specification defining the *data-centric* communication standard for a wide variety of computing environments, ranging from small networked embedded systems up to large-scale information backbones for publish-subscribe data distribution systems. The purpose of the specification is to provide a common application-level interface that clearly defines the data-distribution service. This specification describes the service using UML, thus providing a platform-independent model that can then be mapped into several real platforms and programming languages. The DDS attempts to unify the common practice of several existing implementations [15]. DDS provides a scalable, platform-independent, and location-independent middleware infrastructure to connect *producers* and *consumers*, supporting many QoS properties, such as asynchronous, loosely-coupled, time-sensitive and reliable data distribution at multiple layers (e.g., middleware, operating system, and network). At the core of DDS is the Data-Centric

Publish-Subscribe (DCPS) [12] model, which defines standard interfaces enabling applications running on heterogeneous platforms to write/read *samples* to/from a global data space in a net-centric system.

The *sample* can be imagined as some data values, such as the temperature in a certain place, that have to be published periodically. Those values describe a single logical data object, an "instance" in DDS terms, whose state changes over time. DDS needs to understand what that state is and under what circumstances it should be published. To do that, the middleware allows describing the own data types using XML, IDL, or a programmatic API; application stores its state using those types and allowing to the middleware to publish the state. In this way, the state of instance is held within the middleware, which provides also state maintenance or management facilities.

Applications can use the middleware to share information with other applications by declaring their intent to publish data *sample*, which is labeled with one or more *topics*. Similarly, applications can use the middleware functionalities to access *topics* of interest by declaring their intent to become subscribers. The underlying DCPS middleware propagates data *samples* written by publishers into the global data space, where it is disseminated to interested subscribers. The DCPS model decouples the information declaration access intent from the information access, thereby enabling the DDS middleware to support and optimize QoS-enabled communication. The DDS does not address the protocol used by the implementation to exchange messages over transports such as TCP/UDP/IP, so different implementations of DDS will not interoperate with each other unless vendor-specific "bridges" are provided. With the increasing adoption of DDS in large distributed systems, it has been defined a "wire protocol" standard, namely DDSI (DDS Interoperability), that allows DDS implementations from multiple vendors to interoperate. DDSI is capable of taking advantage of the QoS settings configurable by DDS to optimize its use of the underlying transport capabilities. DDSI is described in terms of a Platform Independent Model (PIM) and a set of Platform-Specific Models (PSM). The PIM contains four modules: Structure, Messages, Behavior, and Discovery. The Structure module defines the communication endpoints. The Messages module defines the set of messages that those endpoints can exchange. The Behavior module defines sets of legal interactions (message exchanges) and how they affect the state of the communication endpoints. In other words, the Structure module defines the protocol "actors," the Messages module the set of "grammatical symbols," and the Behavior module the legal grammar and semantics of the different conversations. The Discovery module defines how entities are automatically discovered and configured. In the PIM, the messages are defined in terms of their semantic content. This PIM can then be mapped to various PSMs such as plain UDP or CORBA-events.

The entities involved into DDS architecture (sketched in Figure 1) are described:

- **Domain:** DDS applications send and receive data within a domain, which provides a virtual communication environment for participants having the same domain id;
- **Domain participant**, i.e., an entity that represents a DDS application's participation in a domain;

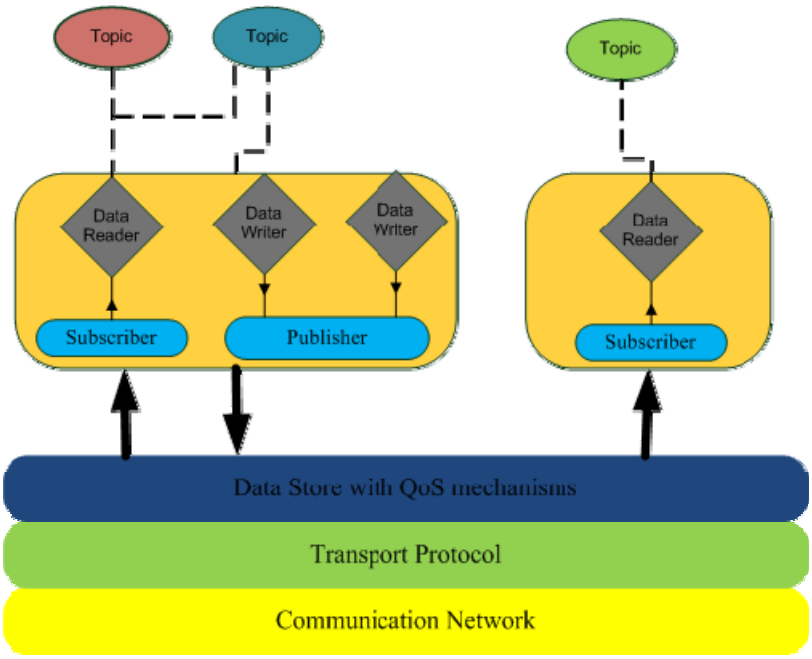


Figure 2 General DDS architecture

- **Data writer and publisher**: Applications use data writers to publish data values to the global data space of a domain. A publisher is created by a domain participant and used as a factory to create and manage a group of data writers that publish their data in the same logical partition;
- **Subscriber and data reader**: Applications use data readers to receive data. A subscriber is created by a domain participant and used as a factory to create and manage data readers;
- **Topic:** A topic connects a data writer with a data reader, i.e.,, communication does not occur unless the topic published by a data writer matches a topic subscribed by a data reader.

2.2 Java Message Service (JMS)

Differently from DDS specification, which is *data-centric* developed with real-time applications in mind and compliant to multiple platforms, other kinds of conventional pub/sub middleware (such as the CORBA Event Service and the Java Message Service) have nowadays been developing with enterprise messaging in mind (i.e., as *message-centric*).

The data-centric model used by DDS can be seen as an extension of the *message-centric* model. In contrast to middleware based on data-centric, a messaging middleware provides no facilities for state maintenance or management. Instead, the system maintains that state externally to the middleware, and when it changes, it sends *messages* about those state changes. The recipients of those messages then decide if and how to update their own state. Because only the application-level logic “above” the middleware has access to its state and knows when

and how to update it, there's no need for the middleware to understand the contents of messages. Messaging middleware implementations therefore typically don't support content-aware message handling and provide more limited control over delivery contracts than do data distribution middleware implementations.

Java Message Service (JMS) is a middleware allowing the exchange of messages among distributed Java applications. JMS provides a standard and common methods to create, send, and receive messages by a message oriented middleware. The JMS architecture (Figure 3)is realized from four elements: ***JMS clients***, the ***JMS provider***, ***Administered objects***, and ***JMS messages***:

- ***JMS clients*** are applications that encapsulate business logic. JMS clients are written in Java and use the JMS API to send and receive messages. JMS clients can also communicate with non-JMS clients, or Java or non-Java client applications using the native client API instead of the JMS API to send and receive messages.
- ***JMS Provider*** is the message server that a vendor provides to implement the JMS API in addition to other messaging services and functionality necessary in an enterprise messaging system. The messaging server provides the necessary infrastructure services to deliver the JMS messages from one JMS client to another JMS client. These services support message routing and providing message persistence.
- ***Administered Objects*** encapsulate provider-specific configuration information and are created and customized by the provider's administrator using the provider's tool and later used by clients. Administered objects can be seen as a preconfigured JMS objects created by an administrator that the clients use for

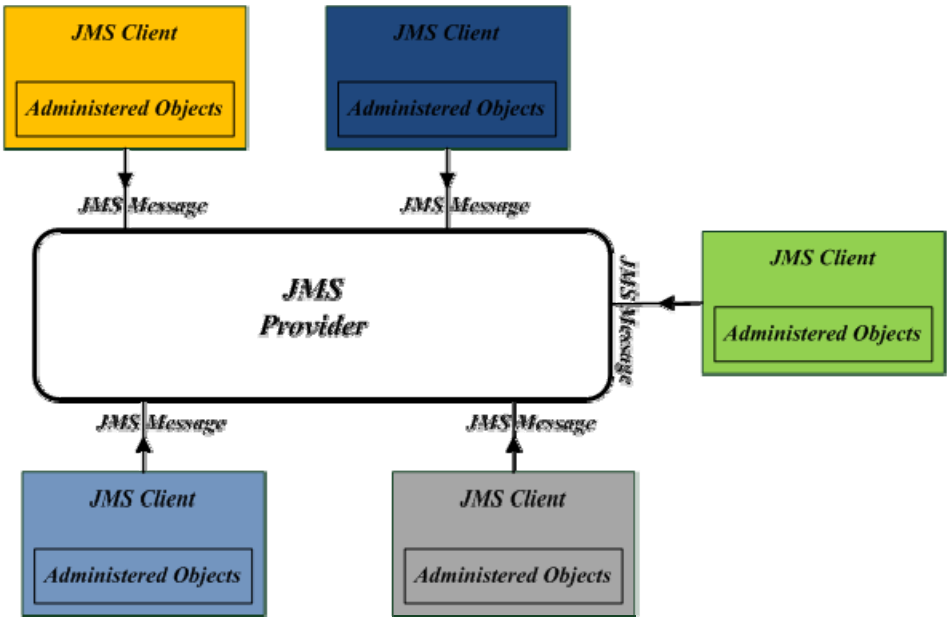


Figure 3 General JMS architecture

providing messaging services. There are two kinds of administered objects: *destination* objects and *connectionFactory* objects. The former is the object that a client uses to specify the destination of a message; in other words, stands for a virtual address. The latter object is the object that a client uses to create a connection with the JMS provider. Destination and *connectionFactory* objects are placed by an administrator in a java naming and directory interface (JNDI) namespace, such as an lightweight directory access protocol (LDAP) directory. The clients use a standard JNDI lookup method to locate these administered objects in a distributed environment.

- ***JMS Message*** defines the message header and the `acknowledge` method used for all messages exchanged among JMS clients.

3. Developing SWIM: SWIM-BOX prototype

The overall system is a grid of SWIM nodes, physically deployed at stakeholders' premises and referred as "legacy" node, which are the actual users of the SWIM common infrastructure. These nodes are allowed to access the SWIM bus through a SWIM-BOX component (see Figure 3). Only SWIM-BOX instances can directly exchange data and invoke services over the net, acting as mediators between legacy nodes and the SWIM-BUS. It is very likely that existing legacy systems are not aware of the SWIM service semantics. Indeed, they could be either built according to different technologies or using different data models, that are not compliant to ICOG. This is the reason why a further software level, named *Adapter*, has been introduced. On the one hand, this provides technology independence, which is one of the SWIM-BOX fundamental requirements. On the other, it

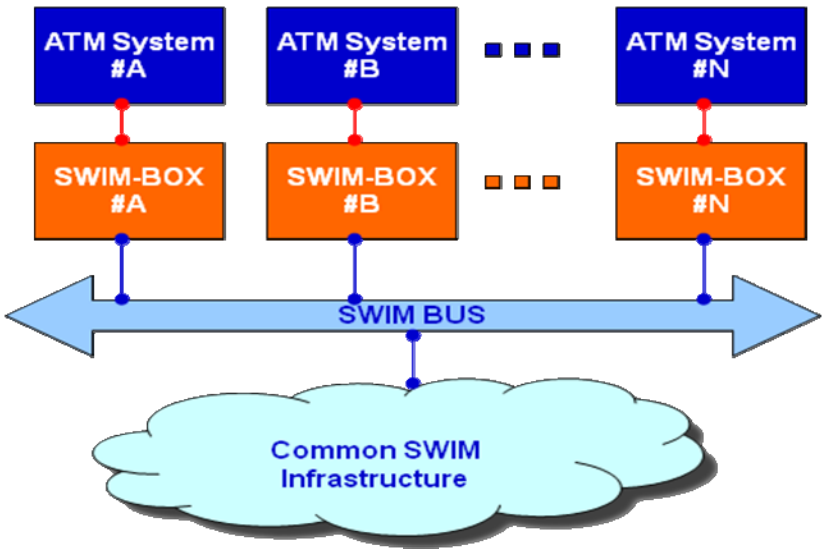


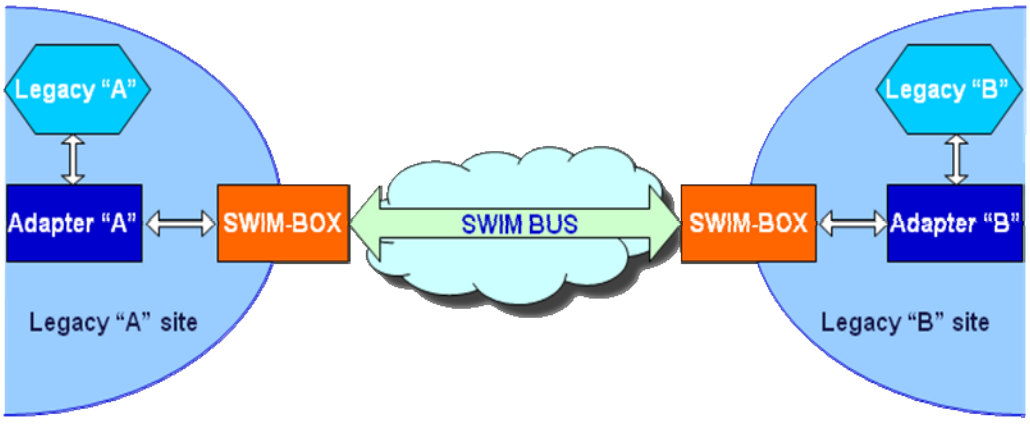
Figure 3. SWIM network architecture

guarantees that all the nodes involved into the network comply with the IGOC standard data model.

High-level architecture from the endpoint perspective is shown in Figure 4, in which the role of adapter nodes is evidenced.

The design and the implementation of the SWIM-BOX have been really challenging, especially from a technological and methodological perspective. For this reason, this work focuses the attention only on the functional aspects of this component to better evidence the lessons learned, as well as the open issues related to its realization. This means that a thorough description of implementation details is out of the scope of this paper.

Figure 4 shows the SWIM-BOX two layered architecture made up of (i) the *Core layer* providing a set of basic and common facilities (e.g., security, data distribution, and registry); and (ii),the *Domain Specific* (DS) layer, in charge of providing domain-related services (e.g., Flight Data Description (FDD), or surveillance subscription, incoming flight notifications).
Data sharing and storage is responsibility of the Shared DataStore (SDS), which filters the client needs of sharing data among SWIM-BOX remote instances. This way, clients are provided with data consistency facility, being unaware of data physical location. SDS, in fact, provides a transparently distributed and transactional storage mechanism allowing them to access and use shared data. The primary duty of this service is to store the data that are not frequently updated, since they need of high availability and automatically synchronized replicas among several distributed SWIM-BOX instances. The SDS has been developed through JBoss Cache [15] implementation, providing a distributed transactional tree cache that can be persistently configured for storing data among a grid of nodes. Indeed, security mechanisms rely on the Security manager component (SEC), which provides support for secure message exchange over the SWIM-SUIT network by implementing authentication, authorization and message confidentiality mechanisms, according to W3C XML Security specifications [17]. SEC provides i) central self-signed certificate management, through a pre-configured key-store containing private keys and certificates associated to all SWIM-BOX instances, and ii) an access control repository, storing user accounts, roles and rules, to enforce authorization policies on client interactions. The Security Manager authorization model takes care of:



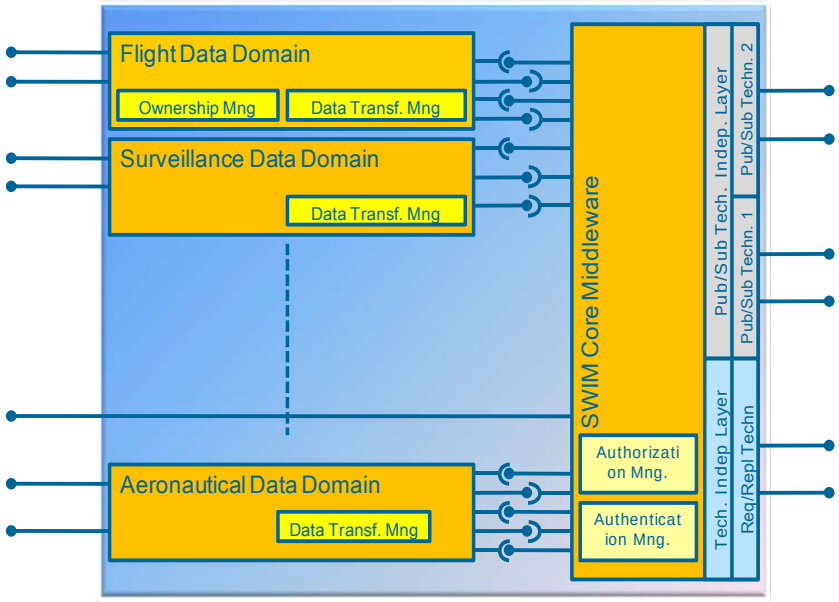


Figure 5 SWIM-BOX architecture

- managing access constraints at service level;
- allowing access to signed and encrypted messages, or portions of them, only to authorized users. Clients are not required to enforce any specific security task (e.g., data encryption), since these are completely managed by SEC, apart from managing certificates for https communication with the local SWIM-BOX instance.

The Publish Subscribe Service (Pub/Sub) component is in charge of distributing the data provided from domain level components by means of the publish/subscribe pattern. In order to assure technology transparency, Pub/Sub actually provides an abstraction layer able to easily substitute the underlying technology without impacting the uppermost domain level components. This has been achieved through the definition of an interface (in order to limit the impact on the performances) that issues the basic operations needed to subscribe and publish data over the SWIM bus. Subscriptions can be requested according to *push* and *pull* paradigms. The former let the subscriber be not blocked waiting for incoming data (asynchronous, push-style). The latter, instead, provides a cache (i.e., pull-point) from which it is possible to retrieve data periodically. Filtering criteria are also provided in this case, both at subscription and at execution time, to select only needed data.

SWIM-BOX prototype exposes two different interface levels, as shown in Figure 5. The first one, stands between the specific domain components and the adapters (i.e.,, at the legacy side) whereas the second one allows interconnecting two SWIM-BOX instances and to let them communicate among each other. Since the envisaged ATM systems are expected to be long lasting, an abstraction layer isolating the actual data distribution middleware technology has been realized, thanks to which the system is currently able to accomplish its tasks (e.g., the data distribution task) transparently, both at the Adapters/Legacy Systems and domain level components sides.

4. Addressed problems

One of the major issues to face in this highly distributed "system of systems" environment is the complexity of the collaborative model induced from the nature of the system itself. The SWIM concept, indeed, primarily aims to pursue interoperability and information sharing among heterogeneous systems and stakeholders exhibiting different requirements and needs, both technical and operational. This nature forces the SWIM providers either to look for standardization, as a primary enabler for interoperability, or to interact with a number of stakeholders having their own background and skills. Since SWIM will be the communication infrastructure on which these heterogeneous systems will be required to interact, it should be able to gain a wider view on the system, assuring the interaction with each stakeholder (e.g., in order to gather requirements). From the stakeholder perspective, instead, such an overall view is not required at all: knowing the ATM business processes in which it is directly involved is more than enough.

From a technical perspective, non functional requirements (such as design for change, scalability, modularity) have been addressed by means of ad hoc design solutions and technological aids. First, the prototype has been designed according to a modular architecture in order to ease the introduction of specialized data domain components, relying on JavaEE EJB3 technology [18]. This allows the components to expose interfaces and services using their own standard and/or technologies. Even though all the components make use of Web Services to expose their interfaces, several standards have been used (e.g., Web Service Notification) to test different solutions. This approach is effective since it allows using different deployment schemas in charge of increasing flexibility and scalability. As an instance, with respect to Figure 3, several deployment options may be provided on a legacy node, depending on the working environment and application workload. One could be "*all-in-one*" solution, in which all data domains components are deployed on a single server and within a single Application server instance, another, completely distributed solution, with each data domain component deployed on a single server and running on a different Application Server instance. Moreover, since the system is expected to operate for a long time, it has been designed to be able to adapt to technological transformations that are likely to occur. The abstraction layer, named Publish/Subscribe Service (Pub/Sub), has been exploited to pursue this goal; it allows providing high degree of robustness to data distribution technology mutations, as well as to support technological diversity over different data domains. Additionally, the Pub/Sub service is aimed to wrap the presence of the different COTS data distribution middleware at lower layer. This is done by providing the data domain components with a single interface which does not depend on the technologies and/or implementations used to perform data distribution. In particular, in the context of SWIM-SUIT, DDS and JMS have been selected to perform data distribution according to the outcome of a formal selection process based on weighted criteria, which have been defined to take into account specific domain requirements.

5. Lessons learned

This section aims to discuss pros and cons of both DDS and JMS technologies, when they are exploited into large-critical systems. Differently from [19] it doesn't aim to provide a performance and technical comparison, rather it wants to highlight what could be done in order to better support developers and system integrators in the task of tuning wide systems for data distribution purposes. Moreover, due to the lack of proper studies for the performance assessment of these systems, the achieved outcomes, in terms of defined metrics, can be taken into account as the starting baseline for the future comparisons.

Actually, DDS provides advanced support to Quality of Service (QoS) and exhibits greater performance benefits. Open source implementation of DDS are just few if compared to JMS. This means that DDS-based solutions are more likely to require expensive license fees and, not less important, they can exploit feedback from a narrow users community. At time of SWIM-BOX prototype implementation only two full OMG standard alternatives, compliant to DDS, were available[20][21].

Conversely, JMS benefits of a broad developers community, which makes newbie programmers initiation easier. Nonetheless, using JMS in advanced configurations settings remains a not trivial task (e.g., we had to use JMS in a clustered configuration) due to the lack of detailed documentation and community experience. On the other hand, DDS is somehow more difficult to start with, since less resources are available on the Internet, but the technology acquisition is rather quick once you have got some familiarity with APIs. The trickiest issues related to DDS arise when different QoS configuration, among the plenty of policies that it is able to provide, must be used.

This is particularly serious when the tuning of wide and complex systems is required. Indeed, when testing the DDS implementation of the Pub/Sub component we experienced a very different behavior over LAN and WAN environments. It is worth noting that this behavior is addressed to DDS complexity into finding the right configuration parameters, because providing more adjustments then JMS, slight variations of them can affect DDS more seriously than JMS.

Stemming from these considerations and due to the nature of the SWIM-BOX prototype, an intensive experimental tests campaign has been carried out with two main technical goals: performance assessment and scalability evaluation.

The former aims to provide the first actual results to be utilized as benchmarking from other frameworks that will come onward. In fact, the proposed overall architecture does not exist yet; legacy nodes exchange data and information through point-to-point communications, as we have already mentioned. To do that, some parameters, such as Operational Throughput (referred as *OT*), and Packets Loss(referred as *PL*), have been measured.

The latter takes into consideration the flexibility of SWIM-BOX infrastructure, fitting to new needs (for instance, in case of new actors interested in the subscription of the same data or information). In fact, the underlying purpose is to understand what happen, in terms of aforementioned metrics, if several nodes request, at the same time, the same resources located

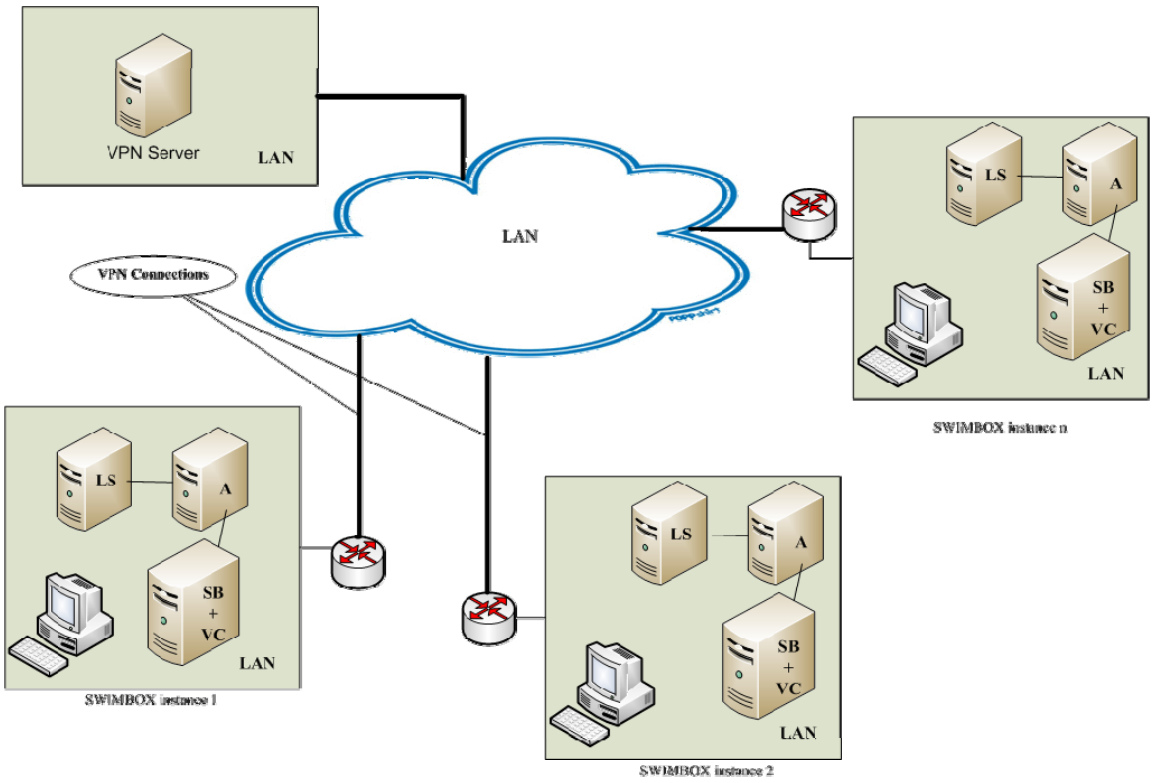


Figure 6 Experimental testbed

on the same SWIM-BOX instance. Is the SWIM-BOX instance capable of managing different operating scenarios?

5.A Experimental testbed

All the measurements have been performed on real test-bed, sketched in Figure 6. It involves different SWIM-BOXs all interconnected through a software VPN built on the top of Internet. No direct data packet exchange is allowed, but all the information are routed to VPN server that than forwards to the proper VPN nodes. On the one hand, such a network organization is very likely to impact on the *quality* and *stability* of achieved results. On the other, it allows getting a "worst case" estimation of the overall picture, i.e, a single centralized VPN server is connected to the LAN and several/heterogeneous links, provided from several stakeholders, are exploited. Indeed, data gathered in such a scenario suffers network load variability influence, as well as packet loss and network delay. Hence, significant improvement in terms of performance can be expected by using a dedicated network with guaranteed level of QoS. Furthermore, according to Figure 4, each SWIM-BOX instance involves a Legacy System (LS), Adapter (A), SWIM-BOX (SB), and VPN client (VC).

As for the Pub/Sub layer, two different types of COTS have been adopted:

- Jboss Messaging (as a JMS COTS and referred in the following as JMS). It has been used in a clustered configuration relying on Jgroups. JMS broker (JBM post office service) utilizes Jgroups UDP multicast messages to exchange synchronisation

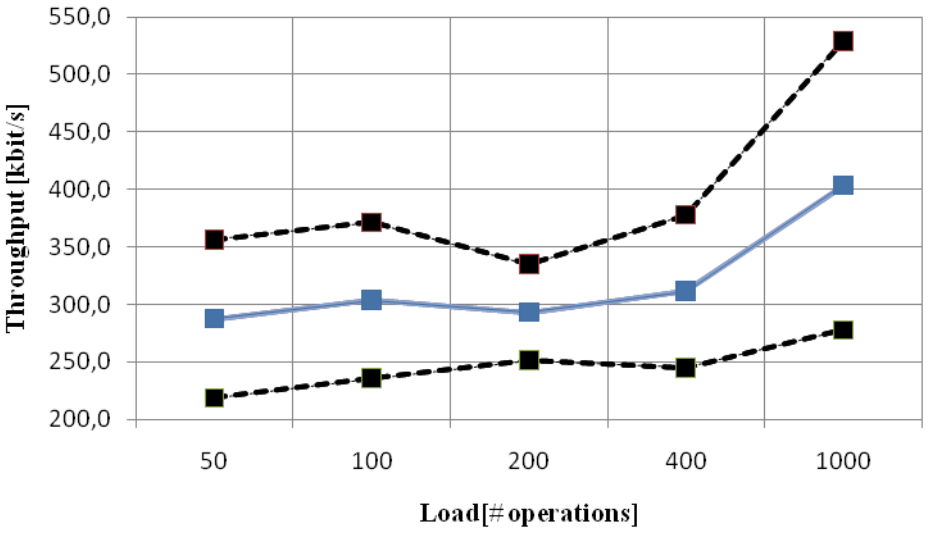


Figure 7 Operational Throughput (OT) due to DDS middleware. The bold line stands for average values, whereas dashed lines defines the standard deviation range.

messages (ControlChannelConfig) and TCP to actually distribute data (DataChannelConfig) across the brokers;

- a commercial DDS, with a suitable configuration in terms of QoS.

5.B Performance assessment

Several tests have been carried out to assess the SWIM-BOX performance. For each test, six measurements have been made and the results in terms of average (i.e.,, μ) and standard deviation (i.e., σ) are provided. For the sake of readability and to make result interpretation

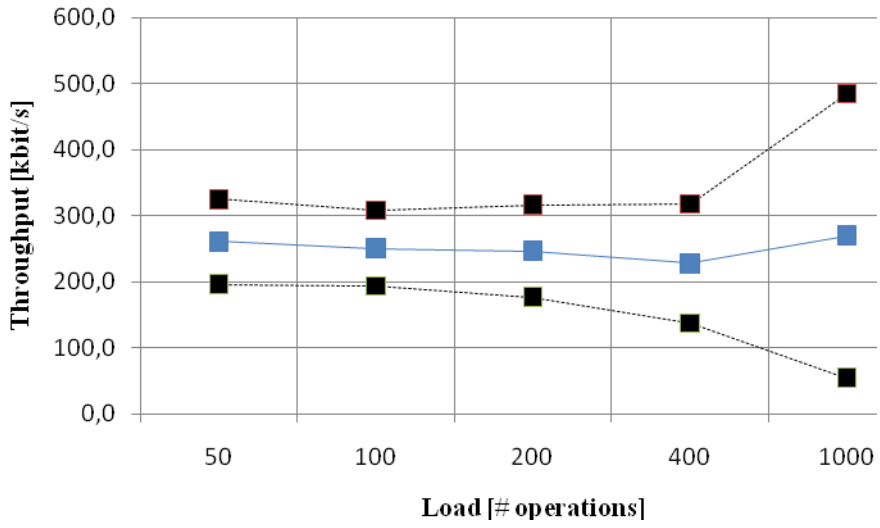


Figure 8 Operational Throughput (OT) due to JMS middleware. The bold line stands for average values, whereas dashed lines defines the standard deviation range.

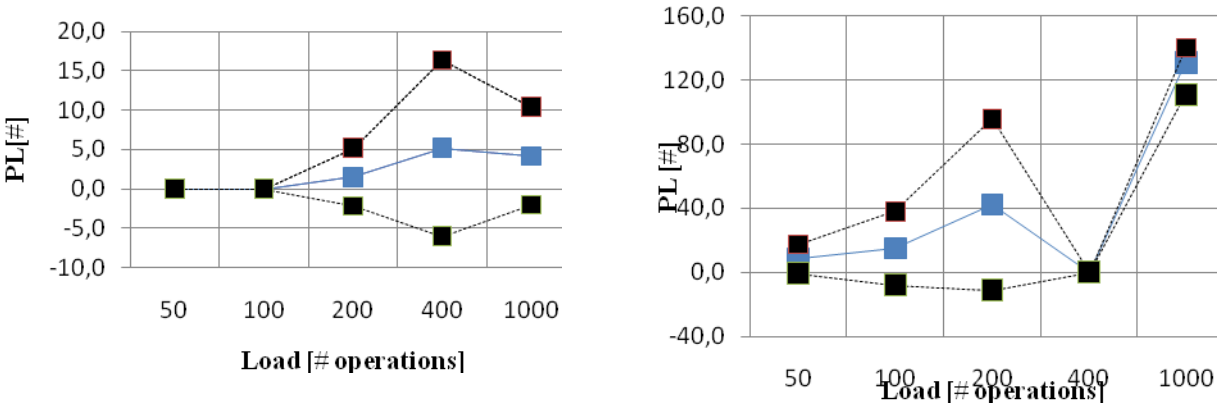


Figure 9 Packets loss (PL) versus number of operations due to a) DDS and b) JMS middleware. In both, bold line stands for average values, whereas dashed lines defines the standard deviation range

easier, only some representative results are reported here. More details can be found into SWIM SUIT deliverables [21]. In particular, in Figure 7 and 8 the OP, respectively for DDS and JMS middleware, are depicted, whereas in Figure 9 the PL versus number of operations is drawn. The continuous lines stand for the average value of each metrics, whereas the dashed lines delimit the standard deviation band.

It's worth noting that the OT average versus the number of operation is almost similar for both the middleware; only slight variations have been observed for number of operations above 400. Further, standard deviation analysis allows claiming that the obtained measurements are i) repeatable in each scenario, due to the obtained σ narrow band, and ii) comparable between the middleware, because the σ bands values are one another overlapped.

On the contrary, different performances have been measured in terms of PL. Indeed, the experienced PL in presence of JMS is higher than DDS. The DDS PL is never above the 20 packets lost, whereas values around 130 packets lost are achieved for JMS.

5.C Scalability evaluation

Scalability tests aimed to evaluate the behavior of SWIM-BOX prototype in presence of multiple connections to the same SWIM BOX instance. In fact, this is the most likely in actual scenario: several actors are allowed and expected to require the same data information to the same provider (i.e., SWIM-BOX instance Figure 10 shows the OT average values versus the number of receivers (i.e., actors invoking the same SWIM-BOX instance) for both JMS and DDS. Some considerations can be drawn:

- JMS and DDS provide the same OT performance; DDS is slightly above of JMS;
- the operational throughput decreases for number of receivers above of five, for both middleware. Above this value, the OT seems to asymptotically reach 300 kbit/s;
- maximum OT is obtained for number of receiver equal to 5 and it value is nearly 360 kbit/s;

Also the packet loss has also been measured, but no relevant behavior has been noticed. In fact, the PL values are never been higher than zero for both the middleware. That is, both

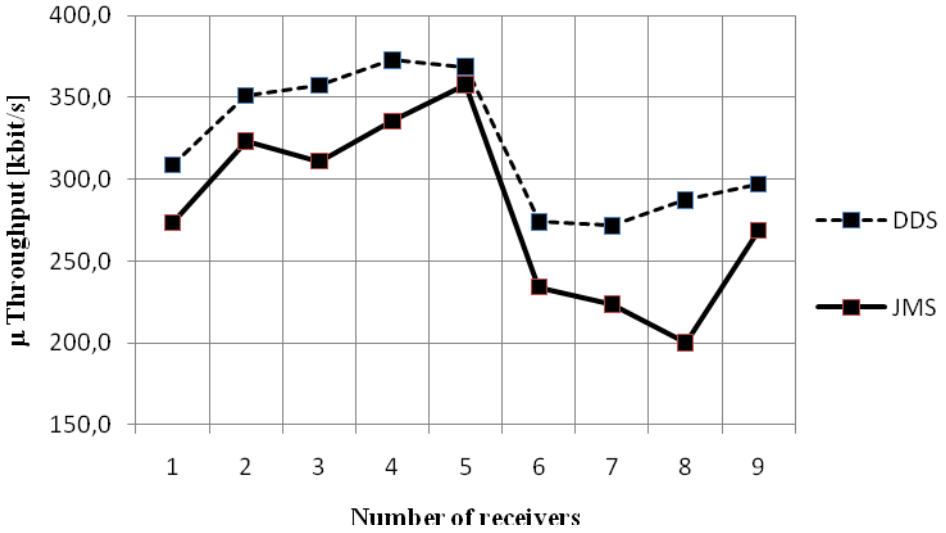


Figure 10 Operational Throughput (OT) versus number of receivers. The continuous and dashed lines stand respectively for JMS and DDS OP average.

DDS and JMS assure a high reliability in term of data packets delivery.

6. Conclusions and ongoing activities

This paper dealt with the experience of implementing a prototypal version of a large scale software system to be used for information sharing in LCCIs. Focusing on the task of data distribution, it highlighted the need for further support tools in charge of making the development and integration tasks easier. In particular, with respect to DDS, the need for a fine tuning tool emerged, able to allow to exploit all the benefits that the technology is potentially able to provide. Otherwise, such benefits would go unexploited, due to the large number of configuration parameters that have to be tuned manually and that make this task difficult and unfriendly. Conducted experiments highlighted that JMS and DDS exhibit different characteristics and performance levels that make them usable in different scenarios, thus allowing to cover a good variety of real network configurations, where number of nodes, load, and geographical distribution of the hosts may change.

The next step aims to exploit and get the most of the broad Open Source community that is currently growing around DDS implementations and technology also in order to assess the interoperability among different implementations of DDS standard. Hence, a new Open Source DDS implementation [23] is going to be integrated into the SWIM-BOX prototype for both measuring its performance and evidencing the presence of interoperability issues.

References

[1] EuroControl Website.

http://www.eurocontrol.int/ses/public/subsite_homepage/homepage.html.
[2] EuroControl. Milestone Deliverable D1 “Air Transport framework: The current Situation. SESAR Library,” www.eurocontrol.int/sesar/, 2006.
[3] Swim project home page: http://www.swim.gov.
[4] Sesar programme homepage: http://www.sesar-ju.eu.
[5] Federal aviation administration: http://www.faa.gov.
[6] ICOG IOP Interface Specification Final Report. 07-May-2008.
[7] ASTERIXCat62 Ed.1.9, http: //www.eurocontrol.int/asterix/gallery/content/public/documents/cat062p9ed19.pdf.
[8] S. Bologna, C. Balducelli, G. Dipoppa, and G. Vicoli, “Dependability and Survivability of Large Complex Critical Infrastructures. Computer Safety, Reliability, and Security,” Lecture Notes in Computer Science, Pp. 342-353, Sept. 2003.
[9] R.M. Dijkman, D.A.C. Quartel, L. Ferreira Pires, and J. Sinderen, “A Design-for-Change Approach: developing distributed applications from enterprise models,” CTIT technical reports series, n. 11, 2002.
[10] Sun Microsystems: Java Message Service, v1.1 SUN Specification, 2002.
[11] Object Management Group. Data Distribution Service (DDS) for Real-Time Systems, v1.2. OMG Document, 2007.
[12] P. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec, ”The Many Faces of Publish/Subscribe,” ACM Computing Surveys (CSUR), Vol. 35, Issue 2, Pp 114–131, Jan. 2003.
[13] The swim suit project: http://www.swim-suit.aero/swimsuit/.
[14] G. P. Castellote, “OMG data distribution service: architectural overview,”, Proc. of Military Communications Conference, Pp. 242-247, 13-16, Oct. 2003.
[15] Gerardo Pardo-Castellote, Stan Schneider, Mark Hamilton, NDDS: The Real-Time Publish-Subscribe Network, Real-Time Innovations, Inc. White Paper, 1999.
[16] Jboss cache project homepage: http://jboss.org/jbosscache/.
[17] W3c xml security working group homepage: http://www.w3.org/2008/xmlsec/.
[18] I. Gorton, A. Liu, “Evaluating the performance of EJB components,” IEEE Internet Computing, n. 4, Pp. 18-23, May 2003.
[19] Rick Warren RTI. From the tactical edge to the enterprise. Integrating DDS and JMS.
[20] http://www.rti.com/products/dds.
[21] http://www.opensplice.com.
[22] http://www.swim-suit.aero/swimsuit/projdoc.php.
[23] OpenSlice DDS Open Source by Prismtech. http://www.opensplice.com/.

Copyright Disclaimer