A Service-Oriented Approach for Network-Centric Data Integration and its Application to Maritime Surveillance

David Parlanti, Federica Paganelli, Dino Giuli, *Senior Member, IEEE*

***Abstract*— Maritime-surveillance operators still demand for an integrated maritime picture better supporting international coordination for their operations, as looked for in the European area. In this area, many data-integration efforts have been interpreted in the past as the problem of designing, building and maintaining huge centralized repositories. Current research activities are instead leveraging service-oriented principles to achieve more flexible and network-centric solutions to systems and data integration. In this direction, this article reports on the design of a SOA platform, the "Service and Application Integration" (SAI) system, targeting novel approaches for legacy data and systems integration in the maritime surveillance domain. We have developed a proof-of-concept of the main system capabilities to assess feasibility of our approach and to evaluate how the SAI middleware architecture can fit application requirements for dynamic data search, aggregation and delivery in the distributed maritime domain.**

***Index Terms*—** maritime surveillance; service-oriented architecture; message-oriented middleware; interoperability.

I. Introduction

Maritime surveillance domain includes all of the activities that can impact the maritime sphere's security, safety, economy and environmental protection. To this end, involved coordination and operational bodies need to share an integrated operational picture for performing their duties in an effective and cost-efficient way. In the European Maritime Policy "Blue Paper" [4], the European Commission states the willingness to "take steps towards a more interoperable surveillance system to bring together existing monitoring and tracking systems used for maritime safety and security, protection of the marine environment, fisheries control, control of external borders and other law enforcement activities". As highlighted in [15], existing systems are based on an "info-centric centralized approach", where a Global Common Operational Picture is built on top of a central data repository system. While such an approach is well suited for addressing the technological interoperability of relatively homogeneous communities, it does not fit well with current requirements for an integrated information picture spanning over technologically and managerially heterogeneous systems. To cope with these requirements, emerging efforts are focused on the adoption of the network-centric approach by leveraging on service oriented architecture models and technological standards.

This work reports on an ongoing study by the Dept. of Electronics and Telecommunications (University of Florence) and by the National Interuniversity Consortium for Telecommunications (CNIT) in collaboration with SELEX Sistemi Integrati. This study aims at investigating the adoption of SOA models and technologies for addressing main information integration requirements of the European maritime surveillance domain. Here we present results in the design of the architecture and on the development of a proof-of-concept for a message- and service-oriented middleware (named SAI – Service Application Integration – system), enabling information search, integration and delivery in the maritime surveillance scenario.

This work is structured as follows: Section II provides a brief introduction to the maritime surveillance application domain and its related interoperability requirements. Section III then presents some relevant high-level and application-independent architectural principles concerning network-centric systems and data integration. Section IV discusses related works on SOA data integration and prepares the reader for the main rationale underlying the SAI system design that is fully presented in Section V. Main implementation choices for the developed SAI Proof of Concept are presented in Section VI, while Section VII reports on the demonstration activities carried out through a special maritime surveillance case study. Finally, Section VIII draws the conclusions and provides some outlines for future work.

II. Maritime surveillance interoperability requirements

"Maritime surveillance" refers to all of the functional areas dealing with assuring maritime safety and security. More specifically, its primary focus includes monitoring, control and enforcement actions for defense, search and rescue activities, maritime traffic control (including anti-terrorism

Manuscript received March 10, 2010, revised June 23, 2010.

D. Parlanti is with the National Interuniversity Consortium for Telecommunications, Florence, Italy, 50139 (e-mail: david.parlanti@ gmail.com).

F. Paganelli is with the National Interuniversity Consortium for Telecommunications, Florence, Italy, 50139 (corresponding author phone: +390554796392; fax: +39055488883; e-mail: federica.paganelli@ unifi.it).

D. Giuli is with the Electronics and Telecommunications Department, University of Florence, Florence, 50139, Italy (e-mail: dino.giuli@unifi.it).

and port security), environment protection, deterrence of illegal goods trafficking, drug trafficking and illegal immigration. In this domain, actors have different duties and responsibilities depending on their institutional role. In the European Union area, which is the reference context for this discussion, most stakeholders may be classified into three categories:

- Inter-National Level Agencies Layer, grouping European agencies in charge of maritime safety and security functions. Examples are: the European Agency for the Management of Operational Cooperation at the External Borders (FRONTEX, http://www.frontex.europa.eu/), the European Maritime Safety Agency (EMSA, http://www.emsa.europa.eu/), the European Police Office (Europol, http://www.europol.europa.eu/).
- Member State Coordination Layer, grouping national level Ministries, such as the Ministry of Environment (MoE), Ministry of Interior (MoI), Ministry of Foreign Affairs (MoFA) and Ministry of Transportation (MoT).
- Member State Operational Layer, grouping Member state Operating Bodies such as Navy, Coastal guard, Port Authorities, Maritime Police and Antifraud corps.

According to current practices, these actors typically collect information for their own purposes by means of dedicated monitoring and surveillance systems, as illustrated in [5]. This situation may lead to many inefficiencies: similar information may be collected by different bodies, while information that could be jointly exploited by several actors is not shareable, just to mention a few examples. In some cases, actors may even be unaware that potentially useful information is being collected by other actors. In most cases, direct information sharing is not possible because of the lack of agreed-on standards and policies.

At present, the European Community is trying to reduce information fragmentation by funding research and experimentation activities targeting the definition of a common information-sharing environment. The objective is to harmonize legal frameworks, to define organization principles for promoting the international and local cooperation and to specify technical frameworks for realizing the information exchange environment across the involved actors. As discussed in [15], the approaches currently adopted for building a common maritime situational picture are typically based on the so-called “info-centric centralized solutions”. In these kinds of systems, an actor has the responsibility of collecting information in a central data repository system to build a global common operational picture and to broadcast it back to the affiliated organizations. While such an approach is well suited for addressing interoperability issues among relatively homogeneous communities, it is does not fit well with the current requirements for building a cross-sectoral integrated information picture through the extension of the information network to a broader community. As a matter of fact, a large amount of information is considered sensitive and its owners are still required to control the extent to which this information is shared with third parties.

Based on this background, our work aims at eliciting and structuring design guidelines for a service-oriented middleware targeting operating environments characterized by a significant technological and managerial heterogeneity, as the one represented by the maritime surveillance domain. To address the needs of such complex and variable interoperability scenarios, our work primarily targets a network-centric approach towards the design of a configurable and extensible middleware offering dynamic data and systems integration capabilities. Companion requirements for security and dependability have also been considered.

Integration requirements may also be elicited while analysing interoperability scenarios within similar application domains, even if with different priorities for the design, development and demonstration activities. As a consequence, the following sections present the SAI middleware by adopting a general-purpose point of view. Demonstration activities carried out in the reference application domain are described in Section VII.

III. Network-Centric Systems and Data Integration

As already stressed by Thomas et al. [38], the network-centric approach mainly focuses on the provision of situational awareness through the sharing of integrated information among surveillance and/or defense bodies. Such a goal has usually been interpreted over the years as the problem of building a centralized repository from data collected through the direct interaction with legacy data-sources. Of course, such an interpretation is more technical than architectural in nature, while presenting at least the following significant drawbacks. The primary drawback is the tight coupling of the solution with the internal structure of the connected data silos. Depending on the application needs, most changes in the data schema of the connected silos could require propagation to the integration system’s “business-logic”, up to make it un-maintainable in the long run in case of frequent revisions. Secondly, the lack of standardized tools and languages for database management would always bind such an integration system to specific technologies, so that technical choices would ultimately drive system design, as opposed to a structured top-down approach. Some concerns also arise regarding the interaction model with legacy data sources. We simply notice here that high-sensitive data that is managed by public institutions typically cannot directly be accessed due to relevant security reasons, thus limiting the number of information sources considered for integration. Finally, information seeding in the maritime surveillance domain usually needs support for asynchronous events notifications. Such a publish/subscribe dispatching model does not fit well with the traditional “pull” database access and querying model. These factors considered, the adoption of the network-centric perspective seems dependent on meeting some basic architectural features highlighted below:

- the coupling of business integration logic with the data’s internal structure should be avoided;
- interaction with legacy data sources should not be based only

on a pull (request/response) RPC-style of interaction but also on a push model (one-to-one, as well as one-to-many);
- the interaction among system components should be based on open standards to avoid technological lock-ins;
- interaction with the integrated data-sources should be mediated by a special indirection-layer for enforcing "on the outside" the legacy system security and data-disclosure policies.

A. Data as a Service: Using SOA for Distributed Data Management

The service-oriented architectural style (SOA) is being widely recognized as an effective approach for achieving the above net-centric requirements while also architecting loosely coupled, event-driven, standardized and maintainable distributed systems [19][28]. Services, the SOA basic system units, are just processes that can be remotely accessed on specific network endpoints. When legacy data is exposed "on the outside" of a legacy system through a SOA service, then that service is truly providing an indirection layer over the legacy system. As a side effect, client applications can now acquire data only by means of message exchange with that service. Since data gathering in the distributed environment is now equivalent to a sequence of service invocations, it turns out that data search, aggregation and reconciliation operations can now be framed as a problem of SOA service composition. By considering data aggregation as a special case of service composition, then SOA mainstream techniques and patterns for service composition, such as those prevailing in the Web Services (WS-*) context, could then be considered as candidate solutions to the problem of data integration among multiple and heterogeneous legacy systems.

IV. Related Work

Among the many available SOA middleware solutions for data integration, our work can be compared with some significant service-oriented systems based on the message-oriented paradigm. "Colombo" is a middleware developed by IBM Research as an experimental SOA platform, based on SOC principles [6]. The Colombo platform is built around the SOAP messaging model and implements most of the Web Service stack, including the WS-* specifications for reliable messaging, security, transactions and service coordination. "WS-Messenger" is a web-service based publish/subscribe system, built on top of a message broker providing reliable message delivery [18]. WS-Messenger is based on WS specifications and supports both WS-Eventing [39] and WS-Notification [22]. "wsBus" is a Web Service-based middleware aiming at supporting dependable WS interactions [8]. It is a lightweight messaging based on the broker pattern which can be plugged into existing Web Services platform to reliably deliver SOAP messages via various transport mechanisms (e.g. HTTP, TCP, JMS).

In the European area, "EuroSur" [10] is an ongoing project providing a common technical framework for promoting and improving the cooperation and communication among member state authorities for improving border surveillance activities. "SafeSeaNet" [9] is a European project focused on supporting information exchange across authorities and operational bodies involved in preventing and detecting illegal pollution actions. SafeSeaNet relies on a centralized messaging system based on XML messages transported over the HTTP protocol.

The "Net-Centric Adapter for Legacy Systems" (NCALS) is a Java software prototype targeting a cost-effective tool for integrating defense legacy systems according to a SOA-based approach [38]. Analogously to our proposal, NCALS applies the adapter, the channel adapter and the message broker patterns to its architecture, but to our knowledge it does not provide any support for secured message exchange, workload distribution or dynamic data-aggregation capabilities.

The remaining systems and architectures seem coupled with the WS-* technology stack and/or with heavy-weight application servers, while also providing just predefined data aggregation workflows. On the opposite, our proposal strives for runtime dynamic data aggregation while still being independent from the WS specification and from commercial or open-source application servers. In this regard, it seems that most interpretations of the SOA approach overlap the service orientation paradigm with Web Services standards and correlated technologies (e.g., the BPEL orchestration language) [33]. Our architecture has been instead driven by high-level SOA architectural principles that have then been translated into practice through the adoption of solid patterns in distributed systems design.

V. The SAI Middleware Architecture

The Service and Application Integration (SAI) middleware architecture is the current snapshot of our research activities on the application of SOA principles to the secured and dynamic aggregation of distributed data from legacy systems. The high-level SAI architecture is shown in Fig. 1. The architecture is conceived as a set of configurable components to be variably assembled into different system deployments to best fit domain-specific integration needs. This implies that concrete middleware deployments are not required to instantiate all of the components envisaged by the logical system architecture. On the contrary, developers can make a number of decisions for targeting the architecture to specific application domains and operating environments. Different specialized deployments of the system are then possible and each of them should be considered just as a distinct instantiation of the same SAI logical framework. In this regard, also system configurability and extensibility are first-class requirements for the architectural specifications, together with the targeted functional capabilities for dynamic data retrieval, reconciliation and aggregation.

The SAI design approach puts service-oriented principles into practice through the adoption of solid patterns in distributed systems design. In software design, patterns provide guidelines for the design of software systems [13].

Since they capture reusable design expertise and solutions to recurrent design problems, they support a higher level of abstraction than implementation-dependent classes and instances [31]. In this direction, the following subsections will provide the rationale for most SAI components in the context of a specific design pattern. Subsections A to I describe SAI components and strategies for legacy-systems integration, service lookup and dynamic data retrieval and composition. Solutions for enabling communication within SAI components and between SAI components and external systems are also considered, while a specific focus is then devoted to the dynamic workload distribution capability provided by the SAI Grid infrastructure. Subsections J to M then analyze how cross-cutting security, dependability and system-consistency concerns have been handled through dedicated components and design patterns.

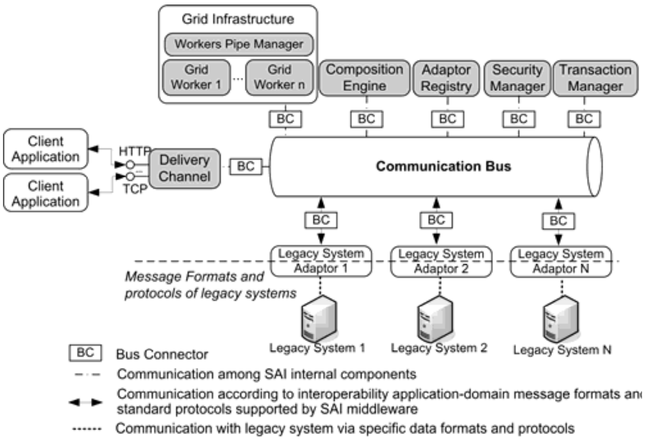


Fig.1 SAI middleware architecture

A. *Legacy Systems Wrappers: The Adaptor Microcontainer*

The goal of the "Adaptor Pattern" is to reconcile legacy components with interface requirements specific to a target framework [1][3]. Reconciliation is usually achieved through a "wrapper" that provides a compliant façade for the legacy interface. The "Adaptor Microcontainer" is the SAI interpretation of the adaptor pattern in the context of an SOA distributed system. The adaptor microcontainer wraps any legacy system into an SAI-compliant message processor. Interfacing with the middleware infrastructure is achieved by connecting the container to the SAI communication bus through a Bus Connector component while adapting the flow of incoming request messages to the proprietary client interfaces of the wrapped legacy system; the response coming from the legacy system is then converted back again into an SAI-compliant message. Depending on the nature of the chosen communication bus, each adaptor can then support both synchronous calls and asynchronous messaging patterns. The internal architecture of the SAI microcontainer is depicted in Fig. 2.

The container acts as a lightweight and configurable message processor hosting a pluggable service implementation that provides the interfacing logic with the legacy information source. In this regard, the typical service implementation will simply interpret received requests into chains of remote invocations to the legacy system interfaces, while leaving additional processing to the container itself. The adaptor micro-container thus manages the life-cycle of the hosted service, while also pre- and post-processing both incoming and outgoing messages through inbound and outbound interceptor chains. By exploiting the "interceptor pattern" [30], the micro-container can so augment the adaptor capabilities while establishing a strict separation of concerns between the purely functional operations performed by the service and the other processing that is possibly required for satisfying specific non-functional requirements. Interceptors are best suited for light processing of several message aspects, including payload compression and message-format adaptations, while allowing transparent adaptor interfacing to the Security Manager and Transaction Manager components. More importantly, interceptors can be "stacked". As an example, let us consider specialized interceptors for message compression and decompression, SOAP envelope document wrapping and unwrapping and security headers processing. A meaningful pre-processing stack would first decompress messages, process SOAP security headers to assess the requestor authentication level and access rights, decrypt the SOAP body, extract the application payload from the SOAP body and then trigger the service implementation processing. Any response from the hosted service would then pass through the same type of interceptors in reverse order to finally obtain a compressed and secured SOAP response message. Of course, security interceptors can also be developed to condition request processing or message dispatching to the authentication and authorization policies of legacy security systems (as depicted in Fig. 2). In this case, security headers would need to be interpreted and translated into remote calls to the legacy security infrastructure, which is just a particular aspect of system integration. This different type of integration, however, would be dealt with by a specialized interceptor, while leaving the basic service implementation free from any security concern. In this sense, the integration logic performed by the hosted service is completely reusable, because it does not depend on the SAI infrastructure, nor on the adoption of specific messaging

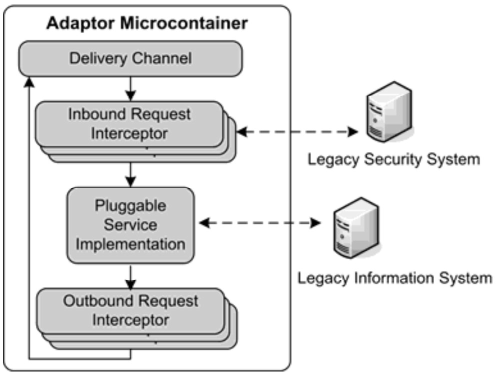

Adaptor Microcontainer
Delivery Channel
Inbound Request Interceptor
Pluggable Service Implementation
Outbound Request Interceptor
Legacy Security System
Legacy Information System

Fig.2 SAI Adaptor

standards.

The set of all active adaptor micro-containers is the lower-level SAI service layer feeding other components and applications with data retrieved by the connected legacy systems. To fit the SAI extensibility goal, adaptors never exchange information with other adaptors: the layer is thus completely modular, with no functional dependency between any two adaptors.

Concerning their deployment, adaptors can be hosted either within or outside of the SAI system boundary. When the adaptor is deployed into the operating environment of a trusted external organization, its control is delegated to the management infrastructure of that organization. In this case, network communication between the adaptor service and the wrapped legacy system is likely to happen under the monitoring and control activities of the legacy system owner. When the adaptor is instead deployed within the SAI system boundary, agreements are required for establishing proper requirements for network communication with the legacy infrastructure. However, specialized interceptors can be used for enabling participation of the adaptor to the transactions possibly activated within the SAI environment, as it is clarified in the next subsection (K) dedicated to the SAI transaction management.

B. Service Lookup: The Adaptor Registry

The "registry pattern" is the commonly accepted design solution for services lookup in distributed environments. The pattern envisages the presence of a logically centralized subsystem supporting lookup capabilities through a "yellow-page" approach. Accordingly, the SAI "Adaptors Registry" manages the "functional profile" of each information system connected to the SAI by means of a dedicated adaptor.

The adaptor profile has to be expressed into an SAI-specific formalism. This requirement arises to ease the integration of systems whose interface description cannot be easily accommodated into the WSDL model and to better support the SAI dynamic data composition capabilities. As in the standard WSDL document-literal binding, SAI profiles model the message-processing capabilities of an adaptor through ordered input-output pairs of XML message types. However, optional "meta properties" can be added to each input-output pair to also describe the non-functional aspects of the transformations performed by the adaptor upon invocation. The SAI profile can also be used to link XML message types to their embedded data atoms. Each data atom is so characterized by its "path", that is by its position inside the containing message structure. Due to the underlying XML information model, data-atom paths always identify either element leaves or element attributes. The final association of atoms with optional "semantic properties" can then be used to link ontological annotations to the data embedded into each message. Fully-populated profiles enable authorized clients to perform complex queries within the Adaptors Registry. More specifically, clients can lookup adaptor endpoints on the communication bus either by target message types, target non-functional properties and/or target semantic annotations. Depending on application needs, the registry can thus support simple UDDI-style service-endpoint retrieval as well as endpoint retrieval by target data atoms or by semantic annotation lookup.

C. Communication Within SAI Components: Bus Connectors

The SAI architecture focuses on the design of reusable, extensible and configurable components. These requirements also mandate the decoupling of every system component from the specific solutions that can possibly be adopted for enabling communication within the same SAI architecture. This need has been satisfied through the introduction of the "Bus Connector" indirection layer over the chosen communication bus. "Bus Connectors" thus enable SAI components to use a common interface for sending and receiving messages to/from the communication bus. In so doing, components become independent from the specific network protocol and/or messaging infrastructure that is chosen for a specific SAI deployment. Since they can intercept both incoming and outgoing messages, Bus Connectors can also be configured for persistent message logging so to support the subsequent analysis of correlated input/output message pairs. As it will be explained in subsection L, such an analysis can be valuable for back-tracking component failures and for supporting state-recovery mechanisms.

D. Communication within SAI Components: Communication Bus

Being a distributed system, interaction among SAI components can only happen through messages: possible interaction patterns between system principals truly depend on the messaging infrastructure that supports internal SAI communications. Thanks to the indirection layer provided by Bus Connectors, choices for the communication bus do not affect the structure of other components, being ultimately dependent just on the target application domain for the architecture. In this regard, it is common practice to bind component communications to the HTTP protocol, especially for its clear semantics and its fit with most enterprise firewall configurations. Use of this protocol usually leads to a request-response, RPC-style interaction model for components. As a side effect of this choice, interacting components become coupled "in time" because they all have to be active during the scope of each request. RPC interactions are so perfectly acceptable for the SAI architecture when applications are not focused on the use of adaptors for the asynchronous provision of data streams, say for near-real-time trend detection for generating alarms. Indeed, use of the RPC interaction pattern in such contexts would eventually lead to continuous component endpoint polling for retrieving relevant events or to continuous connection opening for events-stream provision. The risk here is missing critical events during the polling interval, while also saturating available network-bandwidth, which can occur whenever a lot of data have to be transferred on the wire. As stressed by Helland [16], data-integration domains can also require additional support for message durability and for uniform failover strategies with component failures and restarts. Even the basic need for centralized communication activities monitoring strives for the inclusion of a dedicated messaging infrastructure. When such application-level requirements are relevant, the “Message Broker” pattern envisages the communication among system components to be mediated by a special “broker” component. Such a broker would provide components with a logical “address” (endpoint), while also being responsible for uniform handling of message delivery, publishing, routing and storage. Accordingly, “Message Oriented Middleware” solutions can play the role of such a dedicated broker when the SAI application requirements approach those of an Event-Driven Architecture (EDA) [27].

E. Communication Within SAI Components: Message Format

Messages exchanged over the communication bus among SAI back-end components and adaptors have to be expressed through standardized protocols and language specifications (e.g., XML) as opposed to proprietary binary representations (e.g., serialized objects of a native programming language). The SAI architecture thus satisfies basic software requirements for interoperable remote communications. A typical side effect of this choice is the requirement for marshalling/un-marshalling of received messages into objects of the implementation language of choice prior to their processing by components. In the SAI architecture, this special function can be assigned either directly to the bus connector components or to the pre- and post-processing chains of adaptors micro-containers.

F. Communication Between Applications and The SAI: The Delivery Channel Abstraction

The goal of the “Delivery Channel” component is to provide a unified client interface for structuring client interactions with the SAI architecture. The component is thus designed along the well-known “Façade Pattern”: interactions of clients with the Grid Infrastructure, Adaptors Registry, Adaptors and Composition Engine are factored-out into a minimal set of primitive calls that makes consistent interfacing to most SAI capabilities possible. One of the main goals of these component APIs is to hide the message-creation logic that is required for translating client calls into the interoperable message format internally adopted by the SAI. In this regard, the delivery channel also has to use the specific bus connector associated with the chosen communication bus. Consistently with the structure of the adaptor microcontainer, the delivery channel also supports configurable message pre- and post-processing using dedicated interceptors. In this way, the main client interface can be tailored to the specific choices concerning message format and security schemes of concrete system deployment.

G. Communication Between Adaptors And Legacy Systems

The communication details concerning the remote interaction of an adaptor-hosted service and its managed legacy system has to follow the specific data formats and network protocols that are mandated by the legacy system technical environment. In this sense, they are outside of the SAI architecture’s control and they cannot be customized.

H. Dynamic Data Retrieval and Service Composition

Most of the current strategies for service composition rely on design-time or on request-time definitions of the sequence of service endpoints to interact with in order to obtain target data and process executions. Technologies adopting such an approach include the notable BPEL specification. However, other proposals for workflow languages and implementations still also require clients and/or system developers to feed integration engines with explicit representations of control structures and service endpoints. In terms of system capabilities, defining data-aggregation workflows at design-time would require assumptions concerning a) the number, b) location, c) identity and d) message schema of those adaptor services providing the desired information. Any addition or removal of a legacy-system adaptor would thus compromise the logical correctness of the workflow and the overall functioning of the data integration platform. In complementary terms, request-time defining data-aggregation workflows by clients would force a non-automatic behavior of the system in terms of information retrieval and management capabilities. In this case, clients would be required to communicate to the system: a) where the data services are located, b) the order to follow for information retrieval, and c) how the information should be requested from each data service. The system would play the role of a “proxy” for the external data services, while information search and composition operations would be completely driven by clients, that would be then required to perform the “heavy duty” implied in finding and composing desired information.

To overcome these limitations, the SAI Adaptor Registry and Composition Engine components provide a dedicated infrastructure for semantically handling the adaptors’ functional profiles. Through such an infrastructure, the SAI can reach distributed data atoms through dynamically computed routing tables describing the correct invocation sequence that workflow engines or clients are required to execute in order to “find” requested data in the distributed system. The basic mechanism supporting the SAI service-composition capabilities is outlined below, while forwarding the reader to [25] for a more detailed explanation.

As previously explained, clients can send requests to the SAI system using the Delivery Channel component. Requests for aggregated data have to specify both the “target” message type expected as the SAI-system response and the “input” request information, for example the required search criteria for narrowing the output produced by the system. Upon receiving the request, the SAI system will delegate the request to the Composition Engine component that will then perform the following activities: 1) it will query the adaptor registry for gathering the information required for defining the adaptors invocation sequence (“plan”) that should be followed for retrieving targeted data; 2) it will bind the invocation plan to concrete bus endpoints; 3) it will then execute the solution plan by invoking the selected adaptors and aggregating data embedded into received messages into the target response template specified by the client. The approach followed by the SAI Composition Engine is similar to the AI-planning framework that is described in [41]. More specifically, the SAI interprets data-aggregation as the problem of encoding information expressed by functional profiles into a STRIPS domain. These domains consist of initial conditions that describe the starting state of the world, operators describing the actions that may be performed and goals representing the state to be reached. Operators have preconditions, add effects and delete effects. The SAI Composition Engine assumes initial conditions to be represented by the query input parameters, the goals to be specified by the target output parameters and the operators to be represented by the adaptors functional profile input-output message pairs. For each operation, input messages are modeled as preconditions and output messages as add effects. Moreover, as messages contain inner parameters, as defined in their corresponding XML Schemas, the STRIPS domain is also populated by further operation types representing the syntactic containment data atoms relationships. While existing works such as [41] and [17] aim at optimizing the search algorithm by taking as reference a fixed structure of input and output XML message types, our original contribution is in the extension of the STRIPS approach for accounting the variable and complex structure of the XML messages that can possibly be exchanged in real SOA platforms. By considering the adaptor interfaces as "inference rules", it was then straightforward to apply AI planning techniques to find service composition solution plans [26]. In this direction, we have exploited the Graphplan algorithm that has been first proposed in [2] as an effective way of finding solutions to STRIPS-like domains. As a consequence of this approach, a request for dynamic data aggregation may be handled by the SAI system: i) by invoking a single adaptor service; ii) by executing a composite service, that is by invoking several adaptors in the proper order; iii) by notifying an exception when no known solutions to the planning problem can be provided in the SAI domain. The proposed mechanics for information retrieval and aggregation are proven to be:

- flexible, since invocation workflows are created "on-the-fly" by performing simple operations on the XML messages and atoms graph-representation internally managed by the registry. In this sense, there is no need for request-time or design-time hard-coding of invocation sequences into the system;
- adaptive, since the behavior of information retrieval operations varies according to the number and type of registered services (e. g., legacy system adapters);
- deterministic, since the Graphplan algorithm can provide a definitive response to a user query given the state of the adaptor registry component. The system can so notify users whether a routing table can be computed for achieving target data given user-provided data;
- evolutionary, since more services can be added progressively during the life-cycle of the system.

I. Distributed Computing: The Grid Infrastructure

While focusing on dynamic data retrieval, the SAI architecture also exploits the "Master/Worker" design pattern to provide basic workload distribution to system components and clients. Such a capability can be used for distributed batch processing whenever client applications demand a complex transformation of retrieved data in order to use a single architectural framework both for data access and analysis. Consistently with the Master/Worker pattern, the SAI Grid is based on three entities: a master (the "client" of the grid), a channel for enabling master to worker communication over the chosen communication bus and a set of one or more worker instances. According to the pattern's roles, the master component starts parallelization by defining a set of "jobs" which are then distributed (or "mapped") to worker processes, then waiting for completion of the scheduled tasks. The final step then requires the master to organize (or to "reduce") collected results into a "single" meaningful unit which shall be coherent with the semantics of the distributed work. SAI-specific extensions to such basic workflow include: a) transparent and configurable routing of jobs to workers; b) transparent dynamic jobs reassignment in case of worker failure. The first extension is required for decoupling master implementations from the provision of the needed jobs-scheduling logic. The second extension also decouples the master from continuous job-status monitoring: in case of a worker failure, the infrastructure should then re-assign the job to an active worker with no direct master intervention. To this end, the SAI Grid introduces the "PipesManager" controller component. Masters simply pass their job-scheduling requests to the controller component that then exploits its knowledge on the infrastructure to plan job assignments to specific worker instances. To coordinate the grid infrastructure, the component associates each worker instance to a compound data structure embedding two separate queues for pending and completed jobs. Both queues are monitored by the component to collect statistics on each worker throughput and to detect those failed jobs that need rescheduling. Each worker is notified for new pending jobs, while the PipesManager, once notified of any completed job, notifies in turn the Master component which started the original scheduling request. Since each worker is linked to a unique pipe, job contention is minimized, while routing logic can exploit pipes information to perform scheduling optimization.

J. Cross-Cutting Aspects: Security

Every SAI system operation is scoped within the security context provided by the "Security Manager" component. The Security Manager component enlists authorized principals, including all of the SAI system components, and manages their security credentials and policies while also supporting secured message exchange and enforcement of the Role Based

Access Control model (RBAC) [29] that has been setup by the SAI system administrators. The SAI security framework handles these operations through well-known cryptography techniques, such as encryption and decryption to ensure confidentiality and PKI-based message signing and verification to provide action accountability. The Security Manager public key is so used by principals to verify the identity of the SAI system and to bootstrap a secure tunnel for their initial authentication. Before invoking any system capability, registered principals are indeed required to exhibit their credentials to the security manager component.

The authentication request is a signed object embedding the AES encrypted principal's password and the RSA encrypted AES symmetric key used for password encryption. After successfully verifying the principal's credentials by signature checking, the Security Manager decrypts the AES key and then uses that key to decrypt the password. After successfully verifying the principal name and password combination, the security manager creates a time-limited session-wide RSA public/private key pair for the recognized principal. The session RSA private key is encrypted using the AES key originally specified in the authentication request. The same key is also used for encrypting the retrieved principal roles and authorizations. The new set of credentials is then sent back to the principal by the security manager after signing the response object with its private component key. The principal verifies the source of the authentication response by using the security manager public key. At this stage, the principal is authenticated and can start using the allowed system capabilities. Authenticated principals now possess all the required information for establishing secured communication with the other system components: since the architecture supports message-level security, no other transport level security mechanism is strictly required.

In this regard, it is interesting to sketch how the above security framework interfaces the client delivery channel with the SAI adaptors layer. It has already been shown that both the delivery channel and adaptors can be configured with security support just by adding special processing interceptors in their inbound and/or outbound processing chains. On the client side, security interceptors interface with the security manager to acquire public RSA session certificates and security policies for the target adaptor. Since the client component is already authenticated, its session keys are used with the retrieved information to encrypt and sign the request message payload according to the adaptor's declared security policy. On the adaptor's side, security interceptors ask the security manager for the public session key of the requesting principal, so to assess the identity of the requestor. After a successful check, they can decide whether to proceed with request processing or perform further evaluation by interfacing with the legacy system security infrastructure. In this way, the burden of responsibility is delegated to the legacy system management infrastructure so that legacy systems can always take the final decision concerning whether to allow or not data access to SAI authenticated principals.

K. *Cross-Cutting Aspects: Transaction Management*

Distributed systems can typically achieve state consistency and uniform exception-handling logic through the "Two-Phase Commit" (2PC) protocol [37] and its subsequent enhancements. The 2PC is a centralized protocol where a "Transaction Manager" component is used to coordinate operations over the selected system resources, typically databases and messaging infrastructures. When applied to databases, transactions guarantee atomicity, consistency, isolation and durability (ACID) properties. When applied to compliant messaging infrastructures, no ad-hoc compensation logic is needed when exceptions are thrown during message processing: in these cases, transaction rollback implies both re-delivery of failed requests and canceling of messages that have been scheduled for publishing. Whenever required by component operations, the SAI "Transaction Manager" component can be used for coordinating distributed transactions to ensure the consistency of data access and messaging operations.

Despite its efficacy, use of the 2PC protocol in the SAI is nonetheless conditional on the fulfillment of strict technical requirements. Since interactions with the coordinating Transaction Manager have to be blocking and synchronous, temporal duration of these interactions also has to be predictable and relatively short-lived. Indeed, messaging activities can suffer from strong performance impact due to their scoping within a transaction context, while database resources usually have to be locked until transaction commitment. This interaction model then implies strict control of the system components deployment, service behavior and access policies, for example to guarantee constant data availability and target QoS levels. Of course, such a vertical control is commonly achievable only when the management of the whole SAI system is delegated to a single organizational unit and all adaptor components are deployed within the SAI system boundary.

L. *Cross-Cutting Aspects: Dependability*

System dependability can be defined as "the ability to avoid service failures that are more frequent and more severe than is acceptable" [1]. The SAI architecture achieves dependability by means of: i) the Grid Infrastructure load-balancing and job-failover capabilities; ii) the clustering of adaptor services; iii) the support for basic autonomic capabilities; iv) the Bus Connectors logging capabilities.

Concerning clustering, we distinguish between clustering stateful and stateless adaptor services. Clustering stateful services inherently requires state replication across activated replica. In order to avoid failures due to an inconsistent state, we have decided to first scale stateful services "vertically", that is by running them on dedicated machines, while also starting background threads for replicating the "master" service instance state to secondary ones in order to support a switch-over in case of failure. Clustering stateless services simply requires multiple instantiation of the same system component, together with a load-balancing strategy for dispatching requests. In this regard, some dependencies arise only with respect to the choice for the communication bus component. HTTP clustering should be based on standard load-balancing techniques, usually based on special HTTP proxies or on the tweaking of the DNS configuration, while a MOM communication bus can achieve clustering by simply configuring stateless components to consume messages from the same queue: load-balancing is transparently "coordinated" by the broker, with no additional infrastructural requirements.

The SAI architecture also offers basic autonomicity by supporting the well-known "heartbeat" technique. Heartbeats are control messages that are periodically sent over the communication bus to listening controllers, so that they can be interpreted as special signals by listening components. Hence, if a publish/subscribe messaging model is made available by the communication bus, then heartbeats can allow reaction to change in the overall system state with no need for centralized system monitoring. The SAI adaptors can indeed react when they stop hearing heartbeats from the security components cluster. In that case, adaptors actually react by stopping to accept requests and by stopping sending messages until security heartbeats are resumed. This configurable capability, although very simple to implement, still allows for fail-over strategies that are capable of preserving the security state of the system in a non-trivial manner.

Finally, the SAI can support back-tracking component failures through the Bus Connectors message-logging capabilities. While inspecting correlated input-output message pairs can always be used to track-down a failed-component invocation history, logged messages can also be used to recover the most recent state of stateful components. Of course, this goal can be achieved only when stateful components are "piecewise deterministic", that is when state-change is completely dependent on the content and order of received messages only. In other terms, "piecewise deterministic" components behave "reactively": all of their work is done in response to the trigger of a system component external to the service itself. This property thus makes the implementation of state-recovery mechanisms possible through the sequential re-play of logged messages, as it has been described in [14].

M. Propagating SOA Principles to the SAI Component Level

The SAI architecture propagates its core SOA principles from macro to micro functional levels by specifying guidelines for the internal structure of its components.

At the macro level, we know that the SOA basic system units are represented by services and that services are commonly considered as distributed processors whose behavior can be normatively described by publicly accessible and standardized contracts. We stress here that the term "contract" is almost equivalent to the "interface" concept.

Once we translate this principle into the world of object-oriented design, then the SOA principles can be interpreted at the micro level as the common "design against component interface" advice. Object-oriented systems that follow this advice are inherently modular and extensible because class members are typed according to the required interfaces, while still being internally free from any tight structural coupling with specific implementation classes. Since class members are then bound to a specific interface-implementation class during the object initialization only, for example through constructors or through explicit initialization methods, the resulting object is extensible, meaning that it can change its behavior by simply specifying different implementation bindings for its required interfaces. The SAI then mandates such interfaces-to-implementation binding information (also known as "wiring information") to be "externalizable". This approach implies binding information to be pulled-out from a static class-initialization code and to be put into a separate resource that can be loaded during object initialization. It should be noticed that information on the internal wiring of a component is truly the real configuration of the component. When such a configuration is externalized, then it becomes a resource that can be managed as any other data resource. Externalized wiring information thus allows for the pervasive use of the "Factory pattern" for the runtime injection of class dependencies. More specifically, factory objects can now be generalized so as to load dependency definitions from this type of externalized resource. The application of this pattern in the SAI enables objects wiring into working components to also be driven by administrative signals, that is by messages embedding the desired component configuration. This implies that the SAI architecture can extend administrator capabilities to the runtime definition of the inner structure for its system components. Whenever the internal-component interactions are inherently concurrent and asynchronous, the SAI architecture also suggests the use of an in-memory message bus for centralizing coordination and for monitoring internal threading and messaging activities. Hence, a process similar to the macro-level message-oriented communication-bus is then activated at the micro-level to support both event-driven and message-oriented programming styles. Thanks to these additional requirements, the SAI architecture is characterized by multi-level design consistency. In practical terms, this feature enables system architects to reason over the SAI micro-structure in much the same way they reason over the SAI macro-structure: in most cases, the only real difference is how components communicate with each other. At the macro level, the SAI architecture requires the specification of a communication bus for enabling interaction over the underlying network transport protocol; at the micro level, inter-component communication happens through the process shared memory and it does not require the opening of any network socket. Since both micro and macro levels share the same service-oriented structural and interaction approach, the SAI promotes a "fractal" system thinking that is consistent across all of the architecture layers.

VI. The SAI Middleware Proof-of-Concept

We have developed an SAI Proof-Of-Concept (POC) to carefully assess efforts and implementation problems possibly implied by future prototype-level developments. In this direction, our efforts have focused on main SAI capabilities implementation, while structured component benchmarking and performance optimizations are planned for future improvements towards a full SAI prototype. The goal of the Proof-Of-Concept is then to evaluate the architecture's feasibility needs, especially those concerning the targeted extensibility and configurability capabilities. In the process, we have also tried to assess to what extent current open-source technologies can be successfully adopted in the building of a complex enterprise-level architecture. The main rationale for our POC technological choices are summarized below.

A. The Reference Development Language

The SAI architecture is not targeted towards real-time control capabilities. As a consequence, no special technological constraints arise on the choice of the POC runtime operating system and programming language. To achieve a portable implementation while also taking advantage of solid open-source technologies we have then adopted Java as our reference programming language.

B. The Reference Implementation Framework

The SAI architecture has demanding extensibility and configurability targets. Despite the initial choice for Java as the reference programming language, these requirements together provide compelling reasons for shifting from the Sun's Java EE 5 specification as the reference implementation framework. In developing SAI, Java EE 5 problems are tied both to the specification itself and to its leading implementations, whether commercial or open-source. In its specification, Java EE 5 promotes a development framework that is largely based on the adoption of stateless components in its "service" (EJB) layer. Indeed, the Enterprise Java Beans technology essentially provides RPC or messaging interfaces (Message-driven beans) over equivalent component instances. Of course, to allow for consistent pooling and lifecycle management, component instances are also mandated by the specification to be stateless. J2EE components are then targeted at easing clustering and replication with no standardized support for "singleton" components. On the contrary, support for "singleton" EJBs is available only in selected J2EE containers through implementation-dependent strategies[1] (e.g., the JBoss "Service" code-level annotation). Since the SAI may require the use of stateful services, any use of such proprietary capabilities would eventually make the SAI implementation strictly tied to a specific implementation of the framework, thus limiting fulfillment of its basic extensibility requirement. Though J2EE implementations may differ in their maturity level, we also notice that most of them still rely on the use of heavyweight application servers. Despite its modularity, the J2EE specification and programming model has been progressively tied over the years to complex containers with strong administration, deployment and runtime requirements. By coding against J2EE specs, the risk for the SAI is thus the breaking of its configurability goals and the lock-in to specific J2EE implementations. In order to be as coherent as possible with our goals, we used a different approach for the POC development. We then decided to stick to the basic java "Standard Edition" specification while being supported by light and transparent frameworks to correctly implement the basic component-level patterns devised by the same architecture specification. In this direction, we have widely used the core dependency injection capabilities of the Spring framework [12][32], so to break-down code development into many framework-independent Java interfaces and into class definitions with minimal or zero dependencies on external containers, including the Spring framework itself. The SAI POC interprets the Spring just as a rich and reusable implementation of the Factory pattern: as many technology specialists would say, the codebase is fully "Spring unaware". Indeed, the framework has been adopted just for driving component development along the "everything is service" approach and to externalize dependencies into separate configuration files so to satisfy the SAI component-level configurability requirement.

C. Adopted Open Source Technologies And Libraries

The SAI POC has found valuable support in many open sourced systems and libraries. At present, the Message Bus component is powered by ActiveMQ [35], one of the leading open-source implementations of the JMS specification (Java Message Service, [34]). It has been chosen over other competing MOMs because of its solid message throughput under varying operating conditions and of its configuration flexibility. Being a JMS compliant message bus, it was then possible to directly use the JMS remoting libraries already provided by the Spring framework as Bus Connector implementations. Wiring components to such bus connectors happens through a simple configuration, while still being transparent to the remaining implementation details as mandated by the SAI architecture specification. Current POC adopts the XML format, which is handled internally through the popular JDOM java library. The adaptors functional profiles are currently based on a specific XML Schema. Unpacking functional profiles data atoms into the Adaptor Registry is instead based on the XML Schema Object Model (XSOM) [40] since it is the only general-purpose Java schema parser which we are currently aware of.

The security manager PKI infrastructure was realized using the "Bouncy Castle" open source implementation of the Java Cryptography Architecture (JCA). The current security interceptor implementation, which is used both in pre- and post-processing chains of the adaptor microcontainer and of the Delivery Channel component, is built on top of the WSS4J library [36] and implements WS-Security specifications [20]. The Transaction Manager component is powered by JOTM, which is an open and standalone (container-free) implementation of Sun's JTA specification. The STRIPS planner is based on our refactoring of the PL-PLAN, an open source Java library implementing the Graphplan algorithm [11]. Our extensions have been focused on the caching of

[1] Sun has included in the recent EE 6 specification a standardized "singleton" code-level annotation for EJBs. At the time of POC development, Java EE 6 was just released, with really no working implementations for the specification.

computed plans and of “known-unsolvable” planning problems in order to speed-up client request handling. Finally, the SAI Grid Infrastructure was entirely developed from scratch in the Java programming language by applying patterns and strategies specified by the SAI architecture.

VII. Case study in the maritime surveillance

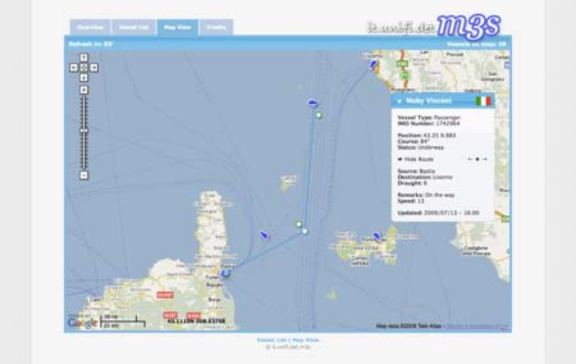
We carried out a case study for a qualitative assessment of the SAI POC model and technologies in the maritime surveillance domain. The preliminary analysis of case-study requirements was performed in the framework of the Operamar European research project in collaboration with SELEX Sistemi Integrati [24]. The case study objective was to perform demonstration activities in a test environment simulating the exchange of basic messages among legacy systems in the maritime surveillance domain. To this purpose, we conceived a basic demonstration scenario aimed at validating most significant features of the SAI middleware.

The demonstration scenario is in the north-Tyrrhenian sea and is composed of two kinds of simulated legacy systems: one providing static information about vessels (e.g. vessel registration and owner details maintained in national registers) and another providing vessel positioning information (i.e. positioning information which might be provided by port authorities or the coastal guard and obtained by vessel position reports, such as AIS messages).

We purposely developed a web application (named M3S: Maritime Surveillance, Security and Safety) providing end users with vessel tracing and tracking services. The M3S Application interaction with data providers (i.e. the simulated legacy systems) is intermediated by the SAI middleware. The demo application allows end users to browse registration information about a set of vessels, selected through query parameters such as country of registration and/or last monitored position. The M3S Web Application also provides a map-based view, showing the current position of monitored vessels (Fig. 3). The user can also access further information on displayed vessels, such as current vessel status (e.g. moored, at anchor, under navigation), route plan and pictures.

We defined a shared XML-based messaging and data model representing a subset of concepts and relations relevant to the maritime surveillance domain. The messaging model is a set of request/response and notification/acknowledge messages. The underlying XML data model includes a core set of data which were chosen based on the analysis of current European practices for sharing data acquired via existing monitoring systems, such as Vessel Monitoring System for Fisheries (VMS), Automatic Identification System (AIS), and Ship Reporting Systems (SRS) [7].

The report of the European Commission on “Legal Aspects of Maritime Monitoring & Surveillance Data” [7] provides a picture of the complexity of data sharing policies in this application domain. According to such legal frameworks and internal policies, information owners may need to specify and enforce specific security policies, e.g. by deciding to disclose information elements only to some organizations and/or in specific circumstances. As described in subsection V.J, SAI adaptor security interceptors can be configured in order to cope with the need of ad-hoc security policy enforcement. Presently the system supports a custom policy language, while future activities are planned to extend the SAI capabilities in order to support standard specifications, such as WS-Policy [21] and WS-Security Policy [23].



Moby Vincent

Vessel Type: Passenger
IMO Number: 1742964

Position: 43.35 9.883
Course: 84°
Status: Underway

Source: Bastia
Destination: Livorno
Draught: 6

Remarks: On the way
Speed: 13

Updated: 2009/07/13 – 16:00

Fig.3 Screenshot of the SAI demonstration web application: map-based view with information about registered vessels

VIII. Conclusions

In this paper we have discussed the SAI approach towards network-centric information sharing and systems integration in the maritime surveillance domain. The SAI approach has focused on the consistent application of SOA principles both at the system and at the component level. The resulting architectural framework is flexible enough to accommodate most of the interoperability requirements implied by the coordination of heterogeneous maritime-surveillance systems and organizations. The developed POC has also demonstrated that SAI implementations can be made free from any technological lock-in, including lock-in to mainstream SOA application servers as well. Hence, we have provided the main rationale for most POC technological choices and we have trialed the POC capabilities through a demonstrator involving common maritime-surveillance application needs.

Regarding future activities, we are fast moving towards a prototype-level implementation of the architecture allowing for middleware performance profile benchmarking under varying deployment configurations. We think that resulting benchmarks will help us design and implement additional infrastructure components and optimizations, so as to achieve state-of-the-art levels of system resilience and scalability. Our research efforts are striving towards two complementary tasks: the optimization of the “Adaptor Registry” semantic-representation capabilities and the evolution of the grid subsystem towards a more flexible cloud infrastructure.

Acknowledgements

This work has been partially carried out in the context of a study funded by SELEX Sistemi Integrati, under the supervision of Agostino Longo. Technical assistance by Luca Capannesi, Department of Electronics and Telecommunications of the University of Florence, is

gratefully acknowledged.

REFERENCES

[1] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, 1, 1, 2004

[2] A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis", Artificial Intelligence, 90 (1997), 281--300

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns (1996), John Wiley & Sons Ltd.

[4] Commission of the European Communities, An Integrated Maritime Policy for the European Union. Communication from the E. C. (2007), http://eur-lex.europa.eu/

[5] Commission of the European Communities. “Towards the integration of maritime surveillance: A common information sharing environment for the EU maritime domain”. Communication from the E. C., (2009), http://eur-lex.europa.eu/

[6] F. Curbera, M.J. Duftler, R. Khalaf, W.A. Nagy, N. Mukhi, S. Weerawarana, Colombo: lightweight middleware for service-oriented computing, IBM System Journal 44, 4 (2005), 799-820.

[7] Directorate-General for Maritime Affairs and Fisheries, European Commission, “Legal Aspects of Maritime Monitoring & Surveillance Data”. Final Report submitted to: DG Maritime Affairs & Fisheries, October 2008, http://ec.europa.eu/maritimeaffairs/studies/legal_aspects_maritime_monitoring_summary_en.pdf

[8] A. Erradi and P. Maheshwari, wsBus: QoS-aware middleware for reliable Web services interactions,", In the Proc. of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service , (2005), 634-639.

[9] European Maritime Safety Agency Extranet, SafeSeaNet, https://extranet.emsa.europa.eu/index.php?option=com_content&task=view&id=70&Itemid=114

[10] EUROSUR, Examining the creation of a European Border Surveillance System, http://europa.eu/rapid/pressReleasesAction.do?reference=MEMO/08/86&format=HTML&aged=0&language=EN

[11] P. Fournier-Viger and L. Lebel, PL-PLAN, Java Open-Source AI Planner, accessed May 2009 http://plplan.philippe-fournier-viger.com/index.html.

[12] M. Fowler, Inversion of Control Containers and the Dependency Injection pattern (2004), http://www.martinfowler.com/articles/injection.html.

[13] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995

[14] V. K. Garg, Concurrent and Distributed Computing in Java. Wiley Interscience, 2004

[15] B. Garnier, J.-M. Lhuissier, A. Guillot, “The global challenge of Information Exchanges in the Maritime Community”, in the Proc. of Maritime Systems and Technology Conference, MAST 2008 (2008).

[16] P. Helland, Data on the Outside vs. Data on the Inside, Microsoft MSDN, Library, http://msdn.microsoft.com/en-us/library/ms954587(loband).aspx

[17] R. Hewett, P. Kijsanayothin, and B. Nguyen, “Scalable Optimized Composition of Web Services with Complexity Analysis”, in Proceedings of the 2009 IEEE international Conference on Web Services. ICWS. IEEE Computer Society, Washington, DC, 389-396

[18] Y. Huang, A. Slominski, C. Herath, and D. Gannon, WS-Messenger: A Web Services-Based Messaging System for Service-Oriented Grid Computing. In Proceedings of the Sixth IEEE international Symposium on Cluster Computing and the Grid (IEEE Computer Society, 2006).

[19] K.B. Laskey, K. Laskey, “Service oriented architecture”, Wiley Interdisciplinary Reviews: Computational Statistics, 1, 1, 101-105, 2009

[20] OASIS Consortium, WS-Security, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[21] OASIS, WS Policy, http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#wspolicy

[22] OASIS, WS-Notification (v1.2), http://docs.oasisopen.org/wsn/2004/06/

[23] OASIS, WS Security Policy, http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html

[24] OPERAMAR Project, “An InterOPERAble Approach to European Union MARitime Security Management”, Final Report for Public Dissemination - Deliverable No4.4, 2008, available online at http://www.operamar.eu/attachments/008_Operamar%20deliverable%20public%20summary%20FINAL.pdf

[25] F. Paganelli, D. Parlanti, D. Giuli. "Message-based Service Brokering and Dynamic Composition in the SAI Middleware", to be included in the Proc. of the 7th IEEE International Conference on Service Computing (SCC 2010), Miami, Florida, 2010.

[26] J. Rao and X. Su. "A Survey of Automated Web Service Composition Methods". In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004, San Diego, California, USA, July 6th, 2004.

[27] A. Rotem-Al-Gaz, Bridging the Gap Between BI and SOA (2007), http://www.infoq.com/articles/BI-and-SOA

[28] A. Rotem-Gal-Oz, What Is SOA Anyway. From Hype To Reality, http://www.rgoarchitects.com/Files/SOADefined.pdf

[29] R. Sandhu, E. Coyne, H. Reinstein and C. Youman, Role-based access control model, IEEE Computer, 29, 2 (1996), 38–47.

[30] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2: Wiley & Sons, 2000

[31] D.C. Schmidt, F. Buschmann, Patterns, frameworks, and middleware: their synergistic relationships, in the Proc. of the 25th International Conference on Software Engineering, (2003), pp. 694-704.

[32] Spring framework. http://www.springsource.org/

[33] M. Stal, Using Architectural Patterns and Blueprints for Service-Oriented Architecture, IEEE Software, 23,2 (2006), 54-61

[34] Sun Microsystems Inc., Java Message Service Version 1.1, http://java.sun.com/products/jms/docs.html

[35] The Apache Software Foundation. ActiveMQ. http://activemq.apache.org/, 2009 ActiveMQ,

[36] The Apache Software Foundation, WSS4j, http://ws.apache.org/ wss4j/

[37] The Open Group, “Distributed TP: The XA Specification”, CN 193, ISBN 1-87263-024-3, 1992. Available: http://www.opengroup.org/bookstore/catalog/c193.htm

[38] A. Thomas, T. Turner, S. Soderlund, Net-Centric Adapter for Legacy Systems, IEEE Systems Journal, 3, 3, 8 (2009).

[39] W3C, “WS-Eventing”, W3C Member Submission (2006), http://www.w3.org/Submission/WS-Eventing/

[40] XSOM, XML Schema Object Model, official web site, https://xsom.dev.java.net/

[41] X. Zheng, and Y. Yan, “An Efficient Syntactic Web Service Composition Algorithm Based on the Planning Graph Model”, In Proceedings of the 2008 IEEE international Conference on Web Services, ICWS 2008. IEEE Computer Society, Washington, DC, (2008) 691-699.