Document Control Sheet

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Originator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-01</td>
<td>Oct 1st, 2009</td>
<td>Initial, including material from Agent Contract Network in collaboration with Munindar Singh and Kartik Tadanki (NCSU)</td>
<td>E. Farcas</td>
</tr>
<tr>
<td>0-02</td>
<td>Oct 29, 2009</td>
<td>Polishing edits</td>
<td>M. Meisinger</td>
</tr>
<tr>
<td>0-03</td>
<td>Oct 29, 2009</td>
<td>Final edits, references</td>
<td>E. Farcas</td>
</tr>
<tr>
<td>1-00</td>
<td>Oct 30, 2009</td>
<td>Added AMQP and DIF models</td>
<td>E. Farcas</td>
</tr>
</tbody>
</table>

Document Information

<table>
<thead>
<tr>
<th>Project</th>
<th>Ocean Observatories Infrastructure Cyberinfrastructure (OOI CI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Custodian</td>
<td>OOI CI Architecture &amp; Design Team (ADT)</td>
</tr>
<tr>
<td></td>
<td>Michael Meisinger (UCSD), OOI CI Senior Architect</td>
</tr>
<tr>
<td>Approval</td>
<td>Alan Chave (WHOI), OOI CI System Engineer</td>
</tr>
<tr>
<td></td>
<td>Matthew Arrott (UCSD), OOI CI Project Manager</td>
</tr>
<tr>
<td>Created on</td>
<td>October 1st, 2009</td>
</tr>
<tr>
<td>Last Changed on</td>
<td>October 30th, 2009</td>
</tr>
<tr>
<td>Document Status</td>
<td>Candidate Draft</td>
</tr>
</tbody>
</table>
# Table of Contents

**DOCUMENT CONTROL SHEET** ........................................................................................................ II

**DOCUMENT INFORMATION** ........................................................................................................... II

1 **INTRODUCTION** .......................................................................................................................... 1

2 **COMMON OPERATING INFRASTRUCTURE** ............................................................................. 1

   2.1 **RICH SERVICE ARCHITECTURE** ......................................................................................... 1

   2.2 **COI FUNCTIONAL COMPONENTS** ....................................................................................... 3

3 **COI GOVERNANCE FRAMEWORK** .......................................................................................... 4

   3.1 **COMMUNITIES AND AGENTS** ............................................................................................ 4

   3.2 **CONVERSATION MANAGEMENT** ....................................................................................... 5

   3.3 **DOMAIN MODEL FOR GOVERNANCE** ................................................................................ 8

      3.3.1 **Overview Governance Model** ...................................................................................... 8

      3.3.2 **Contract Model** ........................................................................................................... 9

   3.4 **GOVERNANCE FOR MESSAGING USE CASE** ................................................................. 10

   3.5 **AGENT CONTRACT NETWORK Prototype** .......................................................................... 10

4 **THE COI MESSAGING SERVICE (EXCHANGE)** ................................................................. 13

   4.1 **EXCHANGE MODEL** ........................................................................................................ 13

   4.2 **MESSAGING SERVICE CLIENT ADAPTER** ....................................................................... 14

   4.3 **AMQP 1.0 PROTOCOL** .................................................................................................... 18

5 **DISTRIBUTED IPC FACILITY** .................................................................................................. 18

   5.1 **DIF MODEL** ....................................................................................................................... 19

   5.2 **DIF AND COI MESSAGING SERVICE** ............................................................................. 20

6 **SUMMARY** ............................................................................................................................... 22

7 **REFERENCES** .......................................................................................................................... 23
1 Introduction

The Common Operating Infrastructure (COI) subsystem is the integrative element of the OOI Cyberinfrastructure. It provides the fabric for other subsystems to integrate existing technologies in form of services that communicate via a reliable message-passing infrastructure. The COI subsystem is a transformative element of the OOI. The COI combines existing production grade software technologies with new software developments. The COI provides:

- A reliable and secure message-based communication network that connects shore-side and wet-side computational environments
- A service-oriented framework to integrate any existing software technology
- Pervasive identity management and security enforcement
- Governance of resources and interactions across multiple domains of authority (facilities)
- A common repository and catalog for any kind of resource under OOI governance

The highest risks associated with the integrative nature and the need for new software development are addressed in the OOI pilot period that followed the OOI Final Design Review in November 2008 and will be ongoing until the end of December 2009. The pilot period’s goals are to prepare for OOI construction and to mitigate significant risks through prototyping.

This report documents the risk mitigation efforts since January 2009 that were relevant for the COI subsystem and their results in the form of a design update for the COI subsystem, as a refinement of the COI FDR baseline architecture. The prototype efforts include the Messaging Service (MS) with its constituent parts broker infrastructure (Rabbit) and service adapter (Magnet), the Agent Contract Network (ACN), and the Distributed IPC facility (DIF).

2 Common Operating Infrastructure

The Common Operating Infrastructure (COI) [14] provides the integration fabric that enables subsystem services to be composed to manage complex interactions. The Messaging Service of COI provides dynamic routing and interception capabilities, a publish-subscribe [11] model for conversations, and reliable storage and delivery of messages to intended recipients across the network.

2.1 Rich Service Architecture

The COI architecture is based on the Rich Services pattern [4], a type of Service-Oriented Architecture (SOA) that provides decoupling between services and allows for hierarchical service composition. As depicted in Figure 1, a Rich Service comprises several entities: (a) the Service/Data Connector, which serves as the sole mechanism for interaction between the Rich Service and its environment, (b) the Messenger and the Router/Interceptor, which together form the communication infrastructure, and (c) the constituent Rich Services connected to the Messenger and Router/Interceptor that encapsulate various application and infrastructure functionalities.

To address service integration, this architecture is organized around a message-based communication infrastructure. The Messenger is responsible for message transmission between communication endpoints. By providing a means for asynchronous messaging, the Messenger supports the decoupling of Rich Services. The Router/Interceptor manages the interception of messages placed on the Messenger and their routing. This is useful for the injection of policies governing the integration of a set of services. The Service/Data Connector encapsulates and hides the internal structure of the connected Rich Service, and
exports only the description and interfaces that the connected Rich Service needs to be visible externally. The communication infrastructure is only aware of the Service/Data Connector, and does not need to know any other information about the internal structure of the Rich Service.

Figure 2 shows the Rich Services pattern applied to the COI architecture; the other five subsystem services networks are encapsulated as Rich Services connected to the COI messaging infrastructure (i.e., the Exchange). This shows the central and integrative role of the COI for the entire Integrated Observatory system-of-systems. The top of the figure depicts the infrastructure services that the COI provides to all subsystems. The COI ensures identity management, pervasive and consistent governance and policy enforcement, state management and resource management. It also enables subsystem services to be composed to handle complex interactions, and manages the overall service orchestration, and enables the presentation of services to the environment. The Router/Interceptor allows for flexible composition between the infrastructure and application services. In this way, there is a clear separation between the business logic and its external constraints. At all abstraction levels, infrastructure services plugged into the Exchange can modify the interaction patterns by re-routing, filtering, or modifying exchanged messages. This feature enables the validation and signing of messages, and the injection of policies governing the integration of a set of services.

The Rich Services integration strategy enables constituent subsystems to evolve independently from the composite system. Subsystem functionality is exposed to the OOI network as services with defined access interfaces, and the only way of interacting within the OOI network is through messages. Service-orientation and messaging realize loose coupling of components, resulting in flexibility and scalability. The complexity of such a large-scale system becomes manageable through separate concentration on each concern. Each subsystem focuses on the services that it enables and assumes that all of the infrastructure services are in place. For example, when designing the Sensing and Acquisition subsystem, the architecture team emphasizes concerns related to instrument control and data acquisition. Instruments can belong to individuals or the marine operators, while all of the deployment platforms are under the marine operator’s authority domain. However, since governance is managed seamlessly by infrastructure services, and can be abstracted when designing the Sensing and Acquisition services, these issues are not of concern to the Sensing and Acquisition service developers.

Each service of Figure 2 is further decomposed according to the Rich Services pattern. For instance, the internal decomposition for the Resource Management services. The Resource Repository service provides references to all resources known to the OOI CI. Through the Resource Integration service, re-
sources can participate in interaction patterns implemented by OOI services (e.g., a storage resource may be used to record states of various services). The Resource Collaboration service provides the collaboration framework between different facilities and the sharing of resources within the OOI federation. The Resource Lifecycle service provides the means to track and manage resources throughout their entire lifecycle from development to decommissioning.

The Rich Services architecture provides resource location independence while user applications are shielded from the complexity of the system and the location of resources. The COI subsystem provides the Resource Management services that enable seamless use of resources across the entire Cyberinfrastructure. Via seamless integration of identity and governance services, the COI architecture supports the deployment, operation, and distributed management of thousands of independently-owned resources of various types (e.g., instruments, processes, numerical models and simulations) across a core infrastructure operated by independent stakeholders, where each stakeholder has different policies.

### 2.2 COI Functional Components

The COI architecture identifies a number of important infrastructure services. The Exchange messaging layer decouples the services of the COI and manages their interplay. The provided infrastructure services include Identity Management, Policy Enforcement, Authentication, Logging, Governance and State Management.

*Governance* defines the policy management framework that is implemented throughout the cyber-infrastructure. The *Identity Management* service provides authentication and supports *Policy Management and Governance*, implementing authorization. It also participates in establishing a Federated Chain of Trust between OOI Facilities as well as components of the CI.

The *Resource Management* Services establish a base for every Resource Management Network in the CI. The *State Management* stores and manages all temporary state information about Identity Management, Policy Enforcement and Ongoing Conversations.

The *Service Framework* stores resources and associates them with their descriptions and relations with other resources. It allows their discovery and subscription.

The *Exchange* service is a fundamental capability of the COI with wide implications on the overall operations of the OOI CI. It implements the message exchange mechanism between the CI services, both within and between services networks. Following the Rich Service pattern, a message-based communications infrastructure manages the service orchestration via two main layers: Messenger and Router/Interceptor. Infrastructure services can modify interactions by re-routing, filtering, or modifying the messages exchanged.
3 COI Governance Framework

The Agent Contract Network Prototype seeks to define the concepts and implementation technologies for agents, contracts, and allied concepts such as policies, which would underlie a network of contracted agents that make up the domains of authority (i.e. facilities) within the OOI system. We illustrate the results of this effort using as example a messaging service conceptualized via Exchange Spaces as communities. The main idea behind our approach is that we express all interactions, especially those corresponding to governance, as arising among autonomous principals. The principals adopt organizational roles to participate in one or more organizations (Orgs). Each Org helps structure the interactions among the principals that feature in it. Each such participation is specified via the contracts that each Org imposes. A specific implementation is a rule-based communicating agent, which stores the applicable rules and information about the state of the world and of ongoing interactions in a knowledge base. We are prototyping such an agent using the Java Agent Development Framework (JADE) and the Java Expert System Shell (JESS).

In the following sections, we present in detail our view of agents and communities, the models for governance, an example, and the prototype implementation in JADE/JESS.

3.1 Communities and Agents

Our approach in distributed computing is based on the premise that independent entities interact in order to pursue shared goals. Entities can represent users, processes, resources and communities.

Entities in the system are represented by their agents. Each entity (or their agent on their behalf) can form any number of relationships with other entities. Relationships are based on mutual (bilateral) agreements between two entities, the results of a successful negotiation. Each entity tracks the consequences (i.e., commitments [9], [16]) of such agreements (i.e., contracts) with other entities. Each observable atomic action of an entity, such as sending a message, that causes a side effect leads to a change and re-evaluation of the aggregate set of commitments of the entity towards other entities.

Entities communicate and collaborate within communities. A community is a specific type of entity in itself. Communities serve multiple purposes in our architecture, including providing a backstop for contracts, providing a locus for naming, and providing a venue to share resources in some uses including infrastructure. A community is represented by a specification that defines the rules for joining the community. Joining a community requires accepting the rules of the community, and the community will provide the registrant entity with a local name and address.

Entities may request to enroll (i.e., participate) in communities or can be invited by other member entities into the community. Enrollment is a symmetric process of negotiation. Entities negotiate the conditions under which they participate in the community and vice versa. If agreement is reached, the resulting contract builds the basis for relations with other community members.

Communities can form relationships with other communities, enabling the members of one community to interact with the members of another community, instituting the specifications of both communities. By contract, the community members are bound to the community specification with its rules, so there is no need for explicit compliance checking (i.e., policy enforcement) and members can interact directly. There might be an imposed requirement for members to leave behind audit trails for later evaluation, same as a tax rule not being directly enforced with every transaction, but which may be audited for compliance to the "state" community tax rules later for each member taxpayer.
We call the set of rules that communities (or other entities) impose policy. Policy to access a resource entity for instance might be an aggregate of many rules, such as the resource owner's rules, the community's rules, and any underlying obligations as consequence of membership.

Figure 3 describes the key ideas of the Agent Contract Network effort in conceptual terms. Each ellipse delineates a community of participants. Communities may be nested into, be disjoint from, or partially overlap with other communities. In the picture above, we see three communities: one called Community A, one called Community B, and one called OOI. The OOI community is the "root" community in that it defines the identities for the parties involved and provides the basic rules of encounter within OOI.

Each community specifies one or more roles. For example, Community A is a resource sharing community. It defines two roles: owner (of a resource) and user (of a resource). The community admits principals who may become a user or an owner (or both). Each owner can contribute its resources to the community, so they can be discovered by any user. A user and owner may negotiate usage terms resulting in appropriate contracts being created between each pair. These contracts govern their interactions regarding the resources they share.

Similarly, Community B models a messaging service realized as an exchange space (inspired by the emerging AMQP standard). This community describes two roles, communicator and distributor. A distributor maps to an exchange point and a communicator to either a publisher or consumer. Each party who adopts a role in this community enters into a contract with the community itself (viewed as a principal in its own right). Each communicator can discover a suitable distributor to publish information to or receive information from.

The OOI acts as an overarching authority. It provides a home for the various application-specific communities that exist within it, and supports the interactions of the principals not only by asserting their identities but potentially by helping monitor and enforce their contracts.

3.2 Conversation Management

Communication between two entities occurs as part of a conversation. A conversation presumes a contract is in place between the two entities intending to converse. This contract must include the common knowledge of an interaction pattern that provides a template for the conversation, with the conversation...
being an instantiation of the pattern. The actual interaction as part of the conversation must comply with the template of the interaction pattern. Each interaction (sending and receipt of a message) potentially causes change in the set of commitments related to the conversation and, thus, indirectly to the commitments between the two entities. Interaction patterns are thereby distributed Assumption/Commitment specifications, in particular also for policy. Each entity can independently monitor the fulfillment of the interaction pattern and contract for the other entity and for itself (and initiate protective or compensating action otherwise). Each party would thus update its commitment store based on each message it sends or receives. Each entity can engage in as many different conversations with different (or the same) entities concurrently as it likes. At any given instant, the effective set of commitments from the point of view of the entity is defined; each interaction can be traced back to a conversation.

We specify interaction patterns using Message Sequence Charts (MSCs, see [7], [9], [12]). We also define a language for commitments that are made and released for each interaction in an interaction pattern. We provide a logical framework to reason over the aggregate set of commitments over time and deduce any implications. Currently, we use a rules engine to implement such a mechanism.

The COI provides collaboration, agreement support, and policy enforcement capabilities. Figure 4 illustrates this pattern for the base case of a single service provider (instrument owner) and consumer (researcher). The pattern generalizes to arbitrary numbers of participants in a service orchestration. Conceptually, the example captures the establishment of a service agreement between two parties; for example, this could unfold between a regional cabled observatory (service provider) and a buoy-based global observatory (service consumer). Each one of the parties has established contractual commitments with their respective user communities, including membership agreements. Upon the establishment of mutual commitments, a contract between the two parties is in place. Further, each party operates under its own set of policies. The negotiation and contracting process, as well as the actual service usage, leads to an interaction pattern between the two parties that is constrained by the contractual commitments and policy declarations of both parties.

Because each Capability Container is equipped with plug-ins for orchestration, governance, policy enforcement, and monitoring/audit, the deployment mapping for the collaboration and policy framework is straightforward: the corresponding interaction interface is stored and accessed CI-wide. Each party’s Capability Container orchestration component executes the projection of the interaction pattern on their respective roles to participate in the overall collaboration. The governance and policy constraints are ex-
The COI, through the use of the CI capability container, factors out the common aspects of communication, state management, execution, governance, and service presentation to provide a highly scalable, secure and extensible model for managing user-defined collections of information and taskable resources. This ability to integrate resources of different types implemented by different technologies is the central value proposition of the architecture. It provides the basis for an integrated observatory network that will remain viable and pertinent over multiple decades.

Protocols are defined through interaction patterns. The interaction pattern (or projection thereof) represents the interaction interfaces of entities (i.e., components). The projection of a protocol on one party can be represented as a Finite State Machine (FSM). We use FSMs as protocol machines that bind the communication endpoint on an asynchronous reliable message-based system to the application logic. Figure 15 shows the use of FSMs as protocol adapters for service applications involved in a conversation as defined by an interaction pattern.

Figure 5 shows an exemplar scenario for the application of agents for the management of physical resources such as sensors, and of services in a distributed environment. Agents interact via the Messaging Service (see Section 4 for details on the Messaging Service). Services themselves use the Messaging Service for inter-service conversations as explained above. In this case, the services’ agents provide the management and control for the service, such as starting/stopping the service and granting access. Finite State Machines as protocol adapters ensure that the agents and service protocols are always in a consistent distributed state, ensuring robustness of the entire system. Service protocol adapters provide access to the service; Managed Resource Agent protocol adapters provide access to the respective resource agents. Resource agents provide monitoring and control of resources, advertise and grant access to resource capabilities and manage the contractual relations and commitments of the resource to its environment on behalf of the resource. All these agent interactions occur in form of conversations based on defined interaction patterns. Proxy Resource Agents provide similar capabilities and interaction patterns but act as...
proxies or supervisors of Managed Resource Agents. Thereby, policy can be applied at various levels within the system through a chain of responsibility.

3.3 Domain Model for Governance

3.3.1 Overview Governance Model

Figure 6 summarizes the key representational and operational concepts of the Agent Contract Network. A Principal is an active OOI entity. A Principal may be an Individual or an Organization (Org). An Organization realizes an Org Specification that states the Contract Templates applying to its various Org Roles. A Principal may play an Org Role, which specifies a Contract Facade consisting of the Qualifications the Principal must meet, the Liabilities it takes on in playing the Org Role, and the Privileges the Org Role grants it. In operational terms, a Principal is represented computationally via a Rule-Based Communicating Agent, which carries out Conversations with other Agents. The Conversations instantiate Interaction Specifications, which aggregate Interaction Patterns specified in terms of Interaction Roles. Each Interaction Role maps to an Org Role and supports its Contract Facade.

This model relates an Org Specification with a Contract. A Contract is specified in Figure 7 to consist of a number of clauses. Each clause of a Contract involves two or more Org Roles. In effect, each Org Role partitions its view of the relevant parts of the Contract. We model the role-relevant parts of each Contract as consisting of three components: qualifications, privileges, and liabilities. In enacting an Org, each Principal that is the actor of an Org Role Participation aggregated within that Org is affected by each of the Roles it adopts. The Principal must be suitably qualified in order to adopt the given Role. By adopting the Org Role, the Principal acquires Privileges (such as powers and authorizations), and becomes subject to various Liabilities (using the term generically to include all manner of commitments where the Principal is a debtor). These requirements on a Principal that are based on the Org Roles it plays are assembled into a Contract Façade.

The Principal applies its (autonomous) Policies, ideally to satisfy its liabilities and take advantage of its privileges. The Principal normally realizes its Contract Façade; not realizing the Contract Façade...
would be a violation. In general, however, we cannot guarantee compliance. There are two main ways to address the question of compliance.

One approach is to be pessimistic and ensure that the actions taken by a Principal are compliant. This is not possible in general since the Principals are autonomous and heterogeneous. However, in cases where we determine the implementation of a Principal, we can place a monitor between the Principal and the rest of the system such that the monitor would allow only the policy-compliant actions of the Principal to proceed.

An alternative approach is to be optimistic wherein we assume the Principals proceed as they would any low-level intervention, but detect and handle noncompliant behavior. This we can accomplish in two ways: either by introducing architectural constructs for monitoring or through the Principals monitoring each other, and potentially escalating matters when there is a problem. Such escalation would be to the Principal that is the Org in whose scope the given Org and its contract exists.

OOI is an Org that serves as the highest scope for the Orgs that we define here. The OOI Org provides identity management as well within this effort.

3.3.2 Contract Model

Figure 7 presents in detail the model for contracts. We model a contract recursively as a set of contracts with the recursion bottoming out as a set of clauses. The recursion is unnecessary in a way but offers a more intuitive representation when compared with real-life contracts where the clauses are structured and the contract thus exhibits a repeating structure.

The clauses in real-life contracts fall into several major categories.

- **The Main Clauses** deal with what the contract is about and the main “business” reason for having a contract in the first place. Naively one can treat a contract as applying between parties that can be viewed as black boxes. However, this is usually not the case in contracts of any importance or complexity.

- **The Normative Clauses** deal with matters that are important to the regulations and policies that apply on the interactions among the parties to the contract. The Normative Clauses are thus of special importance to our proposed use of contracts for governance.

- **The Visibility Clauses** deal with how much access the parties to the contract have to the internal implementations of each other. In general, each party would rely upon such clauses to make sure that the work product is of an adequate quality, that the effort is robust, and does not violate any laws or regulations to which one of the parties might be subject.

- **The Scoping Clauses** specify the purpose and scope of a contract. These are crucial in typical business contracts because of their potential effect of legal rights and such of the parties involved. We expect these might be rather straightforward in most OOI governance settings, although the main OOI membership EULA would have a description of the scoping requirements for when users sign up for an OOI account.

- **The Resolution Clauses** deal with how to respond to failures in a contract, including the possibility of sanctions (of violators) and compensations (by violators). The most likely forms of sanctioning will be through the somewhat amorphous means of reputation and via escalation of complaints to the Org that provides the scope for a contract. The Org may sanction a Principal that it judges to be malfeasant by ejecting such a Principal from the Org and by escalating a complaint further. A malfeasant Principal may be ejected from OOI and declared persona non grata.
3.4 Governance for Messaging Use Case

A first application of the agent contract network concepts was targeted at the COI messaging service (explained below). The goal was to apply contracts and commitments to users of the messaging service. Users of the messaging service include service applications and resource agents. The messaging service itself is a federation comprised of distributed entities that need to keep track of their relationships in form of contracts and commitments. In the following, we explain this case.

Figure 8 identifies the Principals involved in a messaging service, and the governance interactions relating to the publish-subscribe scenario. Its sidebar shows operational interactions that constitute an enrollment negotiation. In the Messaging Service use case, publishing or consumer applications enroll in an Exchange Space (a community) with the purpose of publishing or subscribing to data. Exchange Points within the Exchange Space community play the Org role of Distributors, whereas Consumer and Publishing Applications play the role of simple Communicators.

3.5 Agent Contract Network Prototype

This section describes the prototypical implementation of the basic concepts developed in the agent contract network effort to interactions between entities that belong to different communities.

Java Agent Development Framework (JADE) is a free open-source software framework for developing JAVA Agents. It is distributed under the Lesser GNU License Version 2.1 by Telecom Italia. JADE...
enables us to develop multiagent systems by providing a framework for developing agents and a platform that runs them. The platform itself can span multiple systems that may possibly be running different operating systems. The agents are structured to be FIPA (Foundation for Physical Agents) compliant. Thus performing the basic operations in a FIPA-compliant manner is simple. The ease of use is further enhanced by the graphical tools provided for remotely monitoring, debugging, and managing the lifecycle of agents. With the help of these we can port an agent from one container to another. That is, an agent could be replicated on a different container on the same platform after being terminated on the one it was running on, with no special effort. The framework also implements a few protocols specified by FIPA (like the Contract Net and Subscription among others).

Figure 9 shows the main components and interactions of the agent platform in which the agents are hosted. The agent platform provides a container for the execution of agents, a framework for creating, running, and managing the entire lifecycle of agents, communication infrastructure to enable agent communication, and directory services. Every component of the agent platform runs as an agent. Agents are deployed over the platform and register with the Directory Facilitator to be listed in a accessible Directory. Each agent advertises the kind of services that it provides and thus enables its discovery. Built in agents like the Introspector, Sniffer and Console provide a graphical interface for easy management of the agents.

In JADE, agents are developed by subclassing the jade.core.Agent class. Each agent further has a set of Behaviours (notice the British spellings) that characterize its functioning. All the Behaviours are executed in parallel on the Agent thread. The JADE framework provides several typical agent behaviors, including Cyclic (looping), One-Shot (single run), and more complex ones such as Finite State Machine, which enable the agent to transition from one state to another. The Agent object typically holds many behaviors, mostly in a hierarchy, where a complex behavior such as Sequence contains simpler behaviors. This facilitates a programmer building agents with complex capabilities.

Communication among the agents is enabled by the Agent Management System and is compliant with the ACL (Agent Communication Language) Message syntax as prescribed by the FIPA standard. Support for integrating other transports into JADE is being built. JADE, as was mentioned earlier, provides a set of tools for enhancing the usability, the ease of administration and development.
Remote Management Agent, (RMA), acting as a graphical console for platform management and control. The RMA console can be used, among other things, to manage the life cycle of agents, port them from one container to another and link containers on a platform.

The Dummy Agent is a monitoring and debugging tool, with a graphical user interface and a JADE agent. It is used to send custom messages to other agents and run test conversations.

The Sniffer is an agent that can intercept ACL messages while they are in flight, and displays them graphically using a notation similar to UML sequence diagrams.

The Introspector is an agent that monitors the life cycle of an agent, its exchanged ACL messages and the Behaviours in execution.

The Directory Facilitator is an agent that is launched by default when the container starts and provides the yellow pages service. The user can create a complex network of domains by federating various DF agents appropriately. Each agent is responsible for registering its services with the DF.

JESS is a rules engine implemented in Java and uses the Rete Algorithm for processing rules. It provides a good environment in which to script the Agent rules as not only does it allow for simple representation of rules, but it is also capable of accessing and creating Java Objects. This makes it quite powerful as a rules engine and increases its usability. The Agent uses this framework to reason about which action should be performed next by it. The Agents we build would have the ability to store facts in a Knowledge Base and process rules on them.

We have considered and partially assessed rule representations and technologies. Our current prototype uses Jess. The following are some leading approaches and their pros and cons:

- Jess, the Java Expert System Shell, was originally a reimplementation of the CLIPS expert system shell. Jess supports forward and backward chaining, and integrates well with Java. Its native syntax is based on Lisp, but it supports an XML syntax as well. It is available free for academic use. However, each download comes with usage license for 30 days. A longer term license is possible but imposes onerous terms.

- RuleML is an XML-based representation language for rule systems. It is intended to facilitate interchange among rule systems. Transformations exist between RuleML and other syntax, notably that of Jess. However, RuleML seems not to be in active development anymore and is frozen at version 0.91.
Drools is an open source rules project that supports JSR-94 rules. It is incorporated in JBoss and is supported by other leading open source Enterprise Service Bus implementations such as ServiceMix and Mule. Drools supports only forward chaining. Drools comes with extensive tool support including Eclipse plugins.

Figure 10 shows the main components of an agent implemented using JADE with a knowledge base and reasoner based on the Java Expert System Shell (JESS). JESS maintains and applies the facts and rules for an agent and, thus, enables reasoning and reaction. The Protocol Distribution is a package containing the files with the rules for the various Roles. Each Role is learned by an agent by loading the rules from the respective files. An agent has a JESS Instance in which it stores facts and rules to apply on them. It uses the FIPA compliant communication infrastructure provided by the JADE platform to send and receive messages to and from other agents. Its behavior allows it to invoke Java methods via a Java Callback construct. This enhances JESS its ability to carry out complex tasks on firing a rule.

4 The COI Messaging Service (Exchange)

4.1 Exchange Model

This section describes the fundamental architecture of the COI Messaging Service architecture, as investigated and refined in three prototypes, described below. The three prototypes include the Messaging Service Broker Infrastructure (Rabbit), the Messaging Service Client Application Adapter (Magnet), and the underlying Distributed IPC Facility communication framework (DIF). The following describes the concepts and how they interdepend.

Figure 11 shows two applications interacting. Application here stands for any software client, intending to communicate via the COI Messaging Service. This is the case for all internal service to service interactions, as well as for some external interfaces to the CI. We’ll explain below what qualifications applications have to meet to be able to access the Messaging Service.

The Messaging Service, i.e., the “Exchange”, represents itself to any application as a set of Exchange Spaces. Exchange Spaces are communities in the sense of the Agent Control Network. As such, they provide a community specification, which encodes terms of use. The community’s member entities are the applications using the Exchange Space. Applications have to enroll (register) with the Exchange Space before using its resources, Exchange Points, for message-based communication. Being a member of an Exchange Space enables the applications to interact via message exchange, by producing and consuming messages.

![Application 1](Application 1.png) ![Application 2](Application 2.png) ![Exchange Space](Exchange.png)

Figure 11. Application to application communication scenario

Figure 12 shows a more detailed view of the same scenario. The Exchange Space is represented as a set of Brokers (Message Brokers, for instance AMQP servers [20]) in a distributed networked communication environment. Applications maintain a point of attachment with their Broker, i.e., they maintain a connection to this broker. Different applications may be connected to different Brokers, which all represent the same Exchange Space.

Applications play different roles when interacting with their Broker. They play the roles of Producers or Consumers of messages. In some cases, applications can play both roles. The resources over which Producers and Consumers exchange messages are Exchange Points. Applications can query the Exchange Space for a list of Exchange Points they are interested in, can then request (allocate) the use of an Exchange Point and subsequently produce messages and receive messages from such an Exchange Point.
Receipt of messages is a consequence of a preceding subscription of a Consumer role to an Exchange Point. Note that the concept of an Exchange Point is a logical entity in the Message Service architecture. It is represented everywhere across the distributed Messaging Service.

Internally, the Exchange Space and Exchange Points manage registration of applications, allocation of producers (publishers) and consumers (subscribers) of messages, the efficient routing of messages across the distributed network to consumers, the pre-allocation of message broker resources (exchanges and queues) based on subscriptions transparently to the applications.

The Distributed IPC Facility provides the underlying communication architecture and mechanism for secure interaction with the Messaging Service and within the Messaging Service. Details are explained below.

4.2 Messaging Service Client Adapter

The Exchange (i.e., the COI Messaging Service or the Messenger and Router/Interceptor in the Rich Services architecture) is the central integrating element of the COI. It provides access to the communication mechanisms of Exchange Spaces and Exchange Points throughout the system-of-systems, abstracting from the physical communication infrastructure across multiple domains of authority. Client applications may publish messages on Exchange Points within Exchange Spaces. An Exchange Space represents a “community of interest” that collects and controls all of the Exchange Points in its scope and enforces policy of use for a registered set of users and applications. An Exchange Point is represented through a set of named exchanges on one or multiple AMQP [2] message brokers. Thereby, the Exchange provides a comprehensive, uniform view of a federation of message brokers: from the point of view of a publish/subscribe client (i.e., producers and consumers of messages), the fact that the messaging system is built as a federation of independent message brokers and not as a single broker is hidden.

The CI integration strategy determines how individual software components integrate into the system-of-systems through a message-broker integration infrastructure. The communication system of the OOI CI applies messaging as the central paradigm of inter-application information exchange, realizing the Messaging Service, the integrating element of all services.

Message-oriented middleware (MOM) (see [6], [9]) is based on the concept of a message as the exclusive means of information exchange between the distributed components of a system. All information that is passed between two components or services is contained in messages exchanged asynchronously (i.e., non-blocking) over a communication infrastructure. The sender of a message does not wait for the message to be delivered or returned; it only waits for the MOM to acknowledge receipt of the message. Delivering messages to recipients utilizes the concept of queues. An application component in a message-oriented architecture only knows the incoming queues that it receives messages from as well as the outgoing queues it delivers messages to, plus the message formats that pertain to these queues. The MOM pro-
vides the capability for system integrators to connect these queues to known endpoints (i.e., addresses) in the network; consequently it manages routing, reliable storage and delivery of messages to intended recipients across the network. Standardization is on the way for the underlying message wire transport protocol: the Advanced Message Queuing Protocol (AMQP) [2] defines the interactions of a message broker with its clients, promising interoperability between message brokers of different provenance.

Figure 13 depicts the fundamentals of the CI Messaging Service, as explained above. Message brokers are the central infrastructure elements, which present access to the distributed Exchange Points to all clients. As part of their infrastructure responsibilities, they perform routing and delivery of messages. Message Clients provide the interfaces to the application logic.

![Figure 13. OOI CI Messaging Service application interface](image)

Figure 13. OOI CI Messaging Service application interface

Figure 14 provides an exemplar application scenario within the OOI CI. Capability containers host the application logic that interconnects using the Messing Service. This is exemplified through an Instrument Agent publishing a raw data stream on an Exchange Point (a queue) via messaging. Any number of consumers may choose to subscribe to such an exchange point. In the example, the data processing application as well as the data repository will receive the published messages. A data stream is a continuous series of related self-contained messages on a given exchange point. There is a second exchange point for another data product containing processed data that is consumed by an event detector process. The physical deployment of all applications is irrelevant. The Exchange realizes all connectivity.
Figure 14. OOI CI Messaging Service exemplar messaging scenario

Figure 15 depicts exemplar scenarios of how service clients can adapt to the Messaging Service; we have implemented this in current prototypes [15]. Services are identified by name within the Exchange network throughout the entire system-of-systems. Services are part of distributed applications; the distributed service interaction protocol at every (service) endpoint is implemented by a specialized protocol adapter. Such protocol adapters are instantiated for each conversation instance (see below for further details) through protocol factories; the protocol adapters provide the binding element to the actual service application and its functionality. A typical mechanism of implementing protocol adapters is using Finite State Machines (FSM). FSMs represent each distinguishable protocol condition as a separate state, with defined transitions between states when messages are sent or received, leading to very predictable and robust distributed implementations. We have prototyped several interaction styles between service applications, including direct point-to-point interaction, topic based publish/subscribe fan-out queues and worker queues that facilitate reliable load-balanced applications. The Messaging Service hides the fact that service applications are connected to different message brokers that are operated in different domains of authority.
Our prototypical architecture and implementation of the Messaging Service Client Adapter is named Magnet. It is currently implemented in Python [22], based on the Twisted [23] framework. Python and Twisted represent the technology environment of choice for any CI prototypes and services that do not have any other specific language constraints. One of the strengths of Python is that it has a small memory footprint and efficiently interacts as an integration programming language with interfaces to many other programming languages and execution environments. Twisted is a powerful asynchronous event processing architecture with flexible capabilities and interfaces to adapt to diverse communication mechanisms, including HTTP, AMQP messaging and local file access.

Our Magnet implementation is embedded in the Twisted architecture. Application service clients are implemented as pure Twisted services and protocols. There is no dependency of application services on the AMQP based message-broker infrastructure. In fact, it is possible to replace messaging with a TCP or HTTP based communication channel without notice of the service applications. This is true the other way round as well. Leveraging Twisted, we can bind a wealth of existing protocol and service implementations, e.g., an HTTP reverse proxy with no effort to communicate over an AMQP based communication network.

For any given service application, a Protocol Adapter needs to be developed that translates sent and received messages into commands from and to the application. This is the integration effort required to integrate any application into the system. For instance, a web server application that should be controlled through a web server administration service requires a protocol that translates message based commands (start, stop, configure access, define page etc) into reconfiguration commands for the web server, e.g., through executing local shell commands.

The Protocol Factory keeps track of multiple instances of service protocols. Every connection between to application instances will result in a decision by the factory, whether to create a new instance of the protocol (with its own state, for example through a FSM) or to reuse a singleton protocol. Different communication patterns require different factory and protocol strategies.

We have applied our current Magnet implementation successfully to other subsystem prototypes, including the CEI Cloud Provisioning Environment prototype [18] and the DM Data Exchange prototype [19]. Both successfully and impressively demonstrate the flexibility and power that comes from the message based communication architecture. Achieving this result has reduced the technology risk with applying a message based integration strategy to existing distributed applications substantially.
4.3 AMQP 1.0 Protocol

Part of the Pilot Period was to investigate the emerging 1.0 version of the AMQP protocol. AMQP 1.0 provides a different (simplified) set of abstractions compared with 0.8/0.9 versions. In AMQP 1.0 Broker and Client are just two kinds of Applications related with the classical view of MOM. This specification is more generic, and allows for having producers/consumers into brokers; also, apps can also act as brokers for proxying, etc.

Figure 16 shows the overview of AMQP 1.0 protocol. It defines the concept of nodes as peers in a conversation. There are various types of nodes, the most common ones being producer, consumer, queue, and service. Nodes are aggregated into Containers, which are typically implemented as processes in a regular OS. The standard defines the behavior of two types of containers, namely Brokers and Client applications. Nodes communicate through messages carried over unidirectional links. Links operate on top of sessions carrying commands between containers. Messages exchanged along a session may be fragmented into fixed sized units depending on their size. Commands and their body are encapsulated into communication frames. For two nodes from different containers to communicate, the containers must first establish a connection that carries the frames containing the information exchanged by the two nodes. A connection may multiplex multiple sessions with multiple links. More detailed models for AMQP are available in [24].

5 Distributed IPC Facility

We are currently investigating a special case of community called the Distributed Inter-Process Communication Facility (DIF) [5]. A prototype implementation is currently being developed. Entities, representing processes that require inter-process communication (IPC), enroll in this community and are assigned a name valid throughout the community as well as an address that the community uses internally to direct communication. The resources of the community are local endpoints of the DIF, which provide resource allocation (open/close a connection to another named endpoint) and read/write capabilities.
5.1 DIF Model

Figure 17 shows the overview model for DIF [25]: in John Day’s view, networking is inter-process communication (IPC), where each layer implements the same mechanisms but policies are tuned to operate over different ranges of performance. A layer is a distributed IPC facility (DIF). Application processes communicate via a DIF. The IPC processes that make up this facility provide protocols that implement an IPC mechanism and management tasks. Since the IPC layers repeat, the IPC processes within an IPC facility are in turn the application processes requesting service from the IPC layer below.

The Naming model from Figure 18 explains which names/identifiers are internal or external to DIFs. Note that since the communicating elements are application processes, they also have application names. To become a member of a DIF, an IPC process needs to explicitly enroll, i.e., authenticated and assigned an address.
5.2 DIF and OOI Messaging Service

This DIF facility is intended to be the underlying distributed system primitive within the OOI system-of-systems. As is apparent, in conceptual terms, DIFs relate naturally to the notion of communities that we motivated in the foregoing. Other communities will be defined applying similar patterns for other purposes than communication, such as scalable, elastic computing environments, with entities including the requestors of a service and the responding nodes.

The power of the DIF model is that it can be stacked in order to increase scope. One DIF can leverage a lower level DIF for communication purposes and present a DIF facility of larger scope to its member entities. Thereby, the design of how to architect the communities becomes the driving element in the architecture of a distributed system. Any topology and architecture is possible here, exceeding pure layered architectures.

We are applying the DIF model to the COI Messaging Service. Figure 19 shows the logical distributed concept of the Exchange Space, represented by multiple distributed Brokers across the network, applied to local resources of AMQP message broker instances. Applications in the roles of Producers and Consumers of messages communicate with Brokers on the logical level. At the networking level, this comes down to applications using a connection to a message broker instance to publish messages on AMQP Exchanges and to subscribe to AMQP queues in order to consume messages after they arrive. Realizing a distributed Messaging Service, where applications can be connected to different brokers, requires the AMQP broker instances to federate. We are currently prototyping an extension of the RabbitMQ [21] message broker that provides such a federation. Oversimplified, it comes down to relaying messages that are produced on one broker’s local AMQP Exchange to any remote queue that has subscribers for the same topic of messages, as represented by an Exchange Point.
Figure 19 Distributed Exchange Space/Point concepts mapped to AMQP message broker instances

The broker-to-broker communication in a federation needs to be very controlled, secure, based on mutual trust and, at the same time, efficient and resource aware. In the first step of our prototype, we are applying the concepts of the DIF, implemented as local UNIX file system calls, to broker-to-broker communication. Brokers that intend to become part of a federation need to enroll in the Wide Area DIF. Consequently, they receive a unique name with in this DIF that is then made aware to the other members of the DIF. Any broker can now request a communication flow (i.e., a connection) to any other broker. Through this flow, brokers can distribute messages that arrive on local AMQP Exchanges across the network where there are subscribers to these messages. The DIF makes this exchange secure and exclusive to registered, trusted brokers.

Through its inherent abstraction of names from addresses, the DIF provides the basic mechanisms for efficient relaying of messages across the network. A name can be a representative (indirection) for a number of addresses of actual brokers. By defining unique names not only for broker instances but also additionally for each Exchange Point in the system, brokers can send messages to such a name whenever they receive a message from a local Producer application on such an Exchange Point. The DIF hides the complex algorithms that resolve the DIF name for an Exchange Point to the set of addresses of registered brokers that require updates to the Exchange Point because they have subscribed Consumer clients.

The DIF abstraction is very powerful in describing this interaction pattern and provides a fully secure and scalable communication environment. It is dependent on the policy within the Wide Area DIF alone how routing and network resource management are performed.

Figure 20 shows our vision of applying the same DIF implementation not only between brokers (as the Federated DIF), but also between application clients (DAF stands for Distributed Application Facility) and their local broker instance. We assume that organizations will operate their separate AMQP broker (cluster) instance, and all application clients within the domain of authority of this organization will connect to this local broker. Initially the client to broker connection occurs over the local LAN using TCP. The protocol between clients and brokers is defined by the AMQP Transport specification. We envision the same AMQP Transport protocol occurring over an Organizational DIF. As such, it requires the explicit enrollment of all clients and the broker process and provide then a secure local communication environment that additionally hides any network complexity of organization wide routing and performance.
Figure 20 Organizational client-broker and inter-broker federated DIFs

The same figure also shows the distinction of client applications into Worker and Supervisor roles, in addition to them being Producers and Consumers of messages. The notion, strongly based on the Open Telecom Platform (OTP) [17] design principles, is that any worker needs to have a supervisor that controls the worker’s life cycle and restarts it in case of failure. Supervisors themselves can have supervisors, cascading as a tree of processes that all communicate in a distributed environment exclusively by messages.

6 Summary

The Ocean Observatories Initiative faces the enormous challenge of building a cohesive distributed system-of-systems that incorporates a large number of autonomous and heterogeneous systems, deals with instruments and computational resources of a wide range of capabilities, serves the needs of diverse stakeholders, and accommodates change over the timescale of decades. A carefully thought out architecture is key to addressing this challenge. We find that simplicity wins and a few core principles help us organize the OOI properly. These principles include (1) emphasizing loose coupling through message-based interactions; (2) recognizing the autonomy of the participants by modeling them as agents rather than as traditional objects or pure services; (3) identifying repeating structures (as evinced in our choice of Rich Services, Capability Containers, DIFs, and communities); and capturing and making explicit business-level interactions through first-class status for policy and governance.

We have investigated and refined these principles in the various prototypes related to the COI subsystem. This investigation was cause by the need to prepare the documentation and level of developer experience for construction and to mitigate critical risks coming from technology complexity, dependencies, and lack of experience.

Through prototyping, we could significantly reduce the risks associated with the COI subsystem. The Agent Contract Network effort developed the basic foundations for any relation between two entities in the system in the context of a community (synonymous to organization, facility), represented by electronic contracts and commitments. These concepts have been applied to the core COI components of the Messaging Service. In addition, they are represented within the Distributed IPC Facility implementation, as a specific case of a community for the purpose of communication. These examples show that the fundamental concepts could successfully be applied in a real implementation, justifying their pivotal position within the COI defined architecture. The Messaging Service Adapter implementation (Magnet) showed impressively how a simple abstraction of the communication interface lead to substantial simplification in the design, integration and deployment of distributed service applications, as demonstrated in the case of the Cloud Provisioning Environment and the Data Exchange.
7 References


