SOUSA Developer's Guide

DRAFT

09/01/06

Table of Contents

1 Executive Summary

SOUSA is a peer-to-peer middleware framework designed to facilitate rapid development of large-scale distributed applications for information fusion and situational awareness. In this paper, we described the software architecture of SOUSA, and showed how it uses geographical/temporal indexing of information and reciprocal continuous queries based on state-based and control-based services, to provide a cohesive solution to peer-to-peer connectivity in the context of geospatial information fusion.

SOUSA has been used as the infrastructure for an open system, enabling the dynamic addition of consumers (Browsers) and producers (e.g. Raster Services) in disparate administrative domains. The framework has scaled to absorb Entities providing other geospatially indexed data, including weather reports, VRML models, flight data from aircraft, naval vessels and satellites. SOUSA is presently being further developed and tested at the Center for Computational Sciences (U.S. Naval Research Laboratory) within the context of the Large Data Joint Capability Technology Demonstration (LD-JCTD), under the sponsorship of the National Geospatial-intelligence Agency.

Document Outline: In section 2, we describe the design principles behind the SOUSA architecture. Section 3 provides a "Toy Example" to illustrate a small distributed system build using SOUSA. Developers can use SOUSA in its raw form to create Entities or they can use the cooked form, the Interactive Object Development Library (IDOL). IDOL makes it easy to create Entities that automatically initialize themselves from Properties files. In the sections 4 and 5 we describe development process using SOUSA and IDOL, respectively. Finally, in section 6, we describe the implementation of SOUSA and IDOL.

2 Deducing the Design from First Principles

In designing SOUSA, we assumed that the nature of the data and the capabilities/needs of its consumers cannot be determined at the onset. SOUSA is designed to be open, facilitating the transparent assimilation of both new instances and new types of data providers and information consumers over time.

While many peer-to-peer and multi-agent frameworks exist (e.g.JADE [5], COOLAgent [18], HIVE [27], etc.) none are geared towards the specific needs of information synthesis in the context of realtime situational awareness. Some GIS systems which are close to our application area are the Geographic Resources Analysis Support System (GRASS) [7], Open Source Software Image Map (OSSIM) which is an integrated tool for remote sensing, image-processing, and osgPlanet which is a 3D Geospatial Global Viewer built using OSSIM. In addition, numerous systems (e.g. MASSIV [16]) attempt to apply P2P and multi-agent techniques in the related area of distributed virtual reality, respectively. In what follows, we deduce the design of SOUSA from first principles and requirements.

In SOUSA, the atomic software unit representing both the producer and the consumer of information is called an *Entity*. The SOUSA framework provides common facilities supporting high-level connection semantics between Entities, enabling each Entity to take on the role of either *client* and/or *server* on a peer by peer basis. An Entity may be willing to provide services for other Entities within a SOUSA system; such services are seen to fall into two broad categories: those which are based on *state*, and those which are based on *control*. We demonstrate the qualitative difference between the two categories by way of analogy.

Consider the following actions of a telephone company:

S1. mailing an updated ``Yellow Pages'' to a client business each year,

S2. emailing a fraud alert to a customer because their daily phone usage pattern has changed,

S3. presenting a quarterly fiscal report at a stockholders meeting.

In each scenario, the phone company is providing a contractual *state-based service* to clients. The contract specifies that the client(s) will receive updates reflecting any changes in the state of the service provider, whenever certain parameterized criteria are met. In contrast, consider the following actions of the same company:

C1. processing a client's request to place an advertisement in next year's ``Yellow Pages'',

C2. handling a web-based electronic payment from a customer wanting to settle their bill,

C3. executing the purchase of company shares on behalf of an employee through the stock disbursement program.

In each of these scenarios, the phone company is providing a contractual *control-based service* to a client. The contract specifies that the client may initiate one of a set of parameterized requests, whose execution could result in altering the state of the service provider. In order to be flexible enough to support all future applications, SOUSA Entities have the option of providing both state and control-based services to their clients. In what follows, we describe how this is achieved.

2.1 State-Based Services

A state-based service is a subscription which provides client(s) with an appropriate interpretation of some portion of the service provider's state. One important aspect of SOUSA's design involved addressing (I) how the interpretation is specified, and (II) how the interpretation is computed. The examples S1-S3 in the previous section all involve representing a small part of the telephone company's state in a way that is suitable to the client's information needs. A telephone company might have state-based services for businesses, individual customers, and stockholders, and within each of these classes, it might offer many different state-based service products. As such, state-based services are a means to providing efficient distributed caching [6] of service provider state at each client.

Clearly, it is unreasonable to expect that the service provider would be able to anticipate all possible interpretations that might be useful to its clients right at the initial stages of system design. Given the lifespan of the system, new classes of services and new state-based service products would need to be added, and this would require developing new software to construct the necessary reinterpretations of service provider state. It follows that the SOUSA architecture must facilitate the addition of new interpretive modules over the course of both Entity and system lifetimes. We call these modules *ViewInterpreters*.

2.1.1 ViewInterpreters

A ViewInterpreter is a software module which both specifies and implements a scheme by which service provider state can be adapted to a format that is intelligible and useful for a client. We take the position that over time, new adaptations should be specifiable, by service provider and client developers alike.

When a ViewInterpreter is instantiated on behalf of a client and is installed at a service provider, it is granted read-only access to specific portions of the provider's state. The service provider can deny or grant this access on the basis of the client's identity and trustworthiness, following the model in [1]. Presuming that access is granted, the ViewInterpreter receives notifications whenever the observed portion of the provider's state changes. Upon receiving a change notification, the ViewInterpreter determines if the change is significant (e.g. as in [30,31,26]) recomputes its interpretation if necessary, and determines how the changes in the interpretation should be transmitted back to the client. The ViewInterpreter thus implements a continuous query [25,13,24] at the server, on behalf of the client. A ViewInterpreter can be used to achieve the following types of functionality:

Adaptation. The client need not understand the service provider's state in raw form. A ViewInterpreter serves to transform the service provider's state into a form that is consumable by the client.

Filtering. Not all changes in the service provider's state need to be communicated to the client. The ViewInterpreter can serve to filter out or aggregate changes by transmitting updates to the client only when its interpretation of those changes is significant. A ViewInterpreter thus permits control over the fidelity and timeliness of each client's view of adapter server state, providing a tradeoff between quality and communication overhead in a manner similiar to what is proposed in [9,11].

Intelligent Transport. If the ViewInterpreter determines that the change is significant, it can transmit the new interpretation in an intelligent way. For example, it need not re-transmit the entire interpretation--it may opt to send incremental updates [34,10].

Using adaptation, filtering, and intelligent transport, a ViewInterpreter enables a client to maintain a dynamic ``view'' of a portion of the service provider's state, while ensuring that the view is made to have a particular format and fidelity, both in terms of level of detail and timeliness. It follows that ViewInterpreters serve to maintain an adapted cache copy of the service provider state at each client. We call each such cached copy of service provider state a Receptor.

In our example (S2), a customer who subscribes to the fraud alert service need not be provided with information pertaining to all their calls. Rather, only anomalous calling patterns need to be detected, filtered out, and reported in a compelling way.

2.1.2 ***Receptors and State***

Following the model of LINDA [8], JavaSpaces [33], and T-spaces [39] we implement the state of the service provider as a set of serializable fields, each of which is a key-value pair where the key is a character string. Each Receptor (residing at the client) also contains a set of fields, and the ViewInterpreter is responsible for adapting changes in fields of the service provider's state and determining appropriate changes that should be made in the Receptor's fields. We remark that because of adaptation, the fields in a Receptor need not be a subset of the fields that make up the service provider's state. The client then observes and responds to changes in the fields of the Receptor.

To control latency and timeliness, SOUSA insures that the number of messages in flight between a service provider and one of its clients is kept bounded. If the Receptor at a client is unable to process updates from the ViewInterpreter at the server in a timely manner, the queue of messages in flight eventually reaches its bound and the ViewInterpreter throttles back until the queue size decreases. This does not, however, impede the service provider from modifying its own state--rather any changes that to service provider state that are made when the queue is full are dropped and not transmitted to the Receptor. Timeliness of the client's view of current server state is thus given precedence over ensuring that the client witnesses the entire sequence of intermediate state changes.

The set of Receptors that one Entity issues to other Entities should be seen as a set of distributed cached copies of interpretations of the issuing Entity's state. The coherency of this set of cached copies is regulated by the logic of the ViewInterpreters which efficiently transform changes in the

state of the service provider into changes in corresponding Receptor states [20]. Clearly, if several clients desire the same interpretation of the service provider's state, then the computation associated with the interpretive act need take place only once. At the service provider, clients that need the same interpretation of state are bundled together in an object called a Projector.

2.1.3 Projectors and Multicasting

When several clients need the same interpretation of a service provider's state, these clients are bundled together at the server in an object called a Projector. The Projector encapsulates a reliable multicast channel (based on Norm [2]) to the set of Receptors that are housed in the clients, in much the same way as [4]. At the service provider, a single ViewInterpreter is instantiated, which listens for changes in service provider state and uses the Projector to efficiently convey changes in interpretation to the entire set of clients. In our example (S3), all shareholders interested in the company's quarterly report, so this report need not be regenerated for each recipient. The report can be written once, and mailed in bulk to all shareholders.

The situation may arise where a Projector that encapsulates multiple clients is faced with a subset of slow clients that are unable to process updates from the ViewInterpreter in a timely manner. Left unaddressed, such a situation would ultimately negatively impact the performance all clients, since they form a single group in a reliable multicast protocol. SOUSA enables the Projector to export multicast performance metrics about clients, thereby permitting the service provider to expel slow clients from Projectors, or to dynamically regroup clients into Projectors based on their performance metrics.

2.1.4 Dynamic Parameterization of ViewInterpreters

So far, we have seen how clients can efficiently obtain reinterpretations of service provider state. Now we consider the case when a client wishes to obtain a dynamic interpretation, or more specifically, a parametric interpretation in which the parameters can be dynamically adjusted by the client. This design choice balances the power of server and client. A client's view of server state can change for two reasons: (1) a change in the server's state (initiated by the server) or (2) a change in the parameters defining the interpretation (initiated by the client).

In our example (S1), a client who obtains a view of the Yellow Pages may choose to parameterize this view using two words to specify the upper and lower bounds, e.g. everything between ``DA'' and ``DE''. The view that the client obtains may change because of (1) an underlying change to the Yellow Pages database at the server, e.g. due to the addition or removal of entries; or (2) because the client changes the interpretation parameters, e.g. the client changes the upper bound to ``DZ''.

Depending on the semantics of the specific scenario, dynamic parameterized views enable clients to adjust their ``level of detail'', ``quality of service'', or ``region of interest'' of the state-based service. The client effectively maintains a ``continuous query'' with the service provider [25]. SOUSA provides the infrastructure for efficient (incremental) updates being transmitted whenever client parameters or server state change in a way that causes the reinterpretation to deviate significantly.

Together projectors and receptors implement a variant of the model-view-presenter paradigm, with receptors playing the role of views, and projectors playing the role of the presenter [14]. We now

turn to consider control-based services, which are based on the model-view-controller architecture [15,23].

2.2 Control-Based Services

A control-based service is one which enables a client to initiate any one of a set of parameterized requests, which are executed asynchronously at the service provider, and possibly alter service provider's state. Each role that an Entity is willing to assume is called a Service Interface: a collection of closely related tasks or functions. In our implementation, any Java interface can be used as a Service Interface, specifying the signatures of methods that the Entity may be willing to provide for others. Each Entity maintains an entry in a directory, supplying its network address and the Service Interfaces it supports. Each method in a supported Service Interface is implemented by a plug-in module at the service provider. These plug-in modules are called

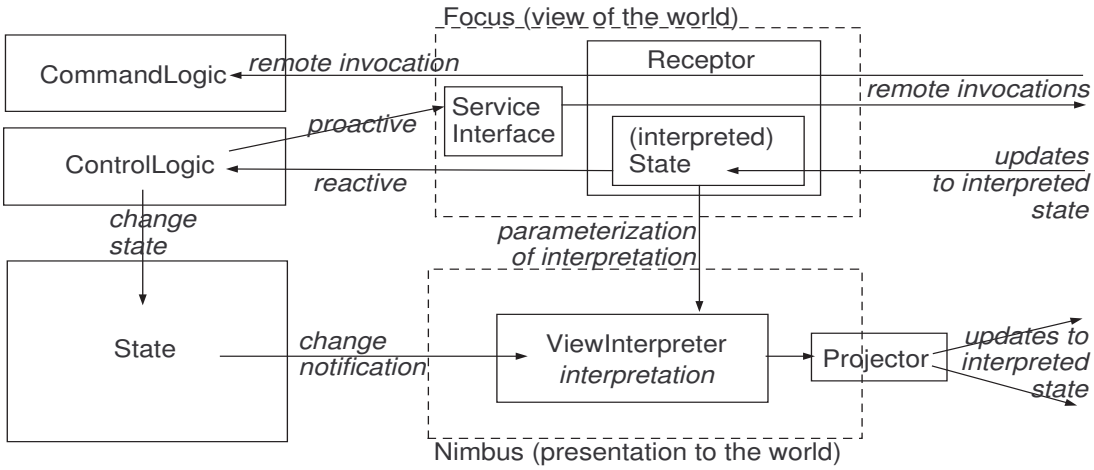
CommandLogic objects.

2.2.1 Asynchronous Remote Method Invocation

When a client invokes a method on the service provider, the invocation should take place asynchronously [29], to avoid unnecessary blocking at the caller. We accomplish this in a manner similar to NINJA [17], JavaParty [28], following the guidelines of [12]. The call returns immediately, providing the caller with a handle. In our implementation, this handle is a SelectableFutureResult object, based on Doug Lea's future pattern [3,22]. The method call itself is reified in a message which is sent from the client to the service provider. There, the method call is routed to the appropriate CommandLogic, resulting in an method execution. The return value is sent by the service provider o the client, where it ``fills in" the future object. Future objects can be bundled together into SelectableSets [21,35], and the client can call select() on such sets. Modelled on the standard UNIX select(), this allows the client to wait for any of a set of concurrent asynchronous remote method calls to complete, enabling high remote execution throughput [19,36].

2.3 Putting It All Together

The various pieces of a SOUSA Entity come together as shown in the figure below. When a client connects to a service provider, it specifies its desired ViewInterpreters and Service Interfaces. Assuming the service provider agrees to the request, a Receptor is returned to the client, and a Java Proxy is made which implements all requested Service Interfaces. This Proxy is backed by an invocation handler which does method call reification and parses return value messages to fill in futures. The specified ViewInterpreters are instantiated and installed at the service provider.



Each Entity collects views of, and provides a view to, other Entities. The collection of views of other Entities is called the Focus, while the collection of views provided to other Entities is called the Nimbus. The Nimbus contains a set of Projectors, which aggregate clients of the Entity based on requested quality of service and client performance metrics. The Focus contains a set of Receptors which reify the Entity's view of state/control based services provided by other Entities. Each Receptor in the Focus is driven by a finite state machine, as is each leaf of a Projector.

The Entity's ControlLogic assumes responsibility for reacting to:

(i) **State changes in the Focus' Receptors.** If a server deems that its state has changed in a way that requires a new interpretation to be projected, the Entity will see this new interpretation as a state change in the Receptor it has received from the server.

(ii) **Membership changes in the Focus and Nimbus**, signaling the arrival or departure of a connection to a client, and/or the completion of a connection to a server Entity.

(iii) **The passage of time.** In the absence of events, the ControlLogic continues to receive opportunities to act.

In response to these events, the ControlLogic may:

(i) **Examine the State of Focus' Receptors.** The Entity may examine the fields of the State of each Receptor, e.g. in response to an indication that a change has occurred.

(ii) **Change the authoritative state of the Entity**. A change in the authoritative state of the Entity results in the notification of all attached ViewInterpreters, which recompute interpretations of the state, for projection to the Entity's clients. The interpretations are parametrized by fields in the Entity's reciprocal view of the clients.

(iii) **Initiate a change in the membership of Focus/Nimbus** (by requesting/dropping connections to server/client Entities). Initiating/dropping a connection to a server Entity, or

dropping an existing connection to a client Entity is an asynchronous operation. The ControlLogic is notified when the requested operation has completed.

(iv) **Initiate a request for control-based services of server Entities to which it is connected.** The ControlLogic may invoke control-based services by calling a method of the proxy which implements the ServiceInterface of a Receptor. When this occurs, the Proxy's invocation handler reifies the method call and sends it for asynchronous execution at the server Entity. On the server, the reified method is interpreted, and results in a method call being executed against a suitable method (having the same signature) from amongsth the server's CommandLogics. The return value of this method call is reified and sent back to the client. The ControlLogic is notified of the return value once the requested operation has completed.

Developing a SOUSA Entity entails developing the following types of plug-in modules: (i) ViewInterpreters, (ii) Service Interfaces, (iii) CommandLogics which implement the Service Interfaces, and (iv) a ControlLogic which responds to connection arrivals and departures, changes in the states of Receptors, the completion of asynchronous method calls, and the passage of time.

3 A Toy Example

This section describes the a toy example to illustrate the interactions among three Entities in a typical scenario. A graphical browser desires to know the imagery that is relevant to its location in space and time. It uses a spatiotemporally aware directory service to learn about other Entities that have relevant information. A spatiotemporally aware raster service advertises with the directory service. When the browser discovers the raster service (through the directory), the raster service delivers imagery relevant to the browser (given its instantaneous spatial and temporal coordinates).

3.1 Directory Server

Every distributed P2P infrastructure must solve the problem of search. Gnutella [37], Napster [38], and CHORD [32] each use their own schemes for indexing information and implementing distributed searching. SOUSA uses a hierarchy of Directory services which index Entities based on the spatiotemporal boundaries of the information they provide.

The Directory defines a control-based Service Interface called Advertiser. This interface specifies methods by which an Entity can advertise itself in the Directory. Specifically, it defines a single method, advertise(), through which advertisers asynchronously supply their contact information, the list of Service Interfaces they offer, and the time-to-live of the advertisement.

The Directory also provides a state-based service using the QueryViewInterpreter. A QueryViewInterpreter delivers filtered directory information to a client, using the client's location as a dynamic parameter. Whenever these parameters change, the QueryViewInterpreter searches the Directory's database for new Entities that provide information near the client's current location and time. It adds newly relevant Entity descriptions and removes old, now-irrelevant Entity descriptions from the advertisements it sends to its client. In order to obtain continually updated search parameters (i.e. the client's current spatiotemporal coordinates), the Directory requests a reciprocal state-based service from its client by specifying a SpatiotemporalViewInterpreter.

The SpatiotemporalViewInterpreter transmits an Entity's notion of its location in space and time along with a set of field names that its client may use to populate the Receptor state it returns to the service. Notice that the Directory is both a service to the Browser (providing directory information about other services) and a client of the Browser (consuming the Browser's location in space and time as inputs to the directory search).

In short, to set up a connection, the Browser requests QueryViewInterpreter from the Directory. The Directory service returns a Receptor that contains the contact addresses of nearby

Entities. The Directory requests a reciprocal connection to the Browser using the SpatiotemporalViewInterpreter. The Browser returns a Receptor that is updated continuously with its spatiotemporal coordinates.

3.2 Raster Service

The Raster Service has a priori knowledge of the Directory. It requests a connection using the Advertiser quality of service from the Directory. The Directory returns a Receptor that implements the Advertiser interface. The raster service invokes the Receptor's advertise() asynchronous RMI method to inform the Directory of the Service Interfaces that it supports and the spatiotemporal extent of the information it manages. The Raster Service then disconnects from the Directory Service. From time to time, it reconnects with the Directory Service to update its information. If the Directory Service does not receive an update within the time-to-live, it removes the Raster Service's advertisement from its contents. Like the Directory, the Raster Service provides the QueryViewInterpreter quality of service to browsers and other clients. QueryViewInterpreter is parameterized by the output of a client's SpatiotemporalViewInterpreter (with the client acting as a server, as described above).

3.3 Browser

The Browser consumes information from the Directory. To obtain directory information, the browser requests QueryViewInterpreter connection from the Directory. The Directory service returns a Receptor that contains the contact addresses of nearby Entities. The Directory requests a reciprocal connection to the Browser using the SpatiotemporalViewInterpreter from the Browser. The browser returns a Receptor that is updated continuously with its spatiotemporal coordinates.

The Browser examines the Receptor obtained from the Directory and selects the Entities that interest it. From each of these Entities, it requests a connection that implements the QueryViewInterpreter quality of service. Each contacted Entity requests a reciprocal connection to the Browser by specifying the SpatiotemporalViewInterpeter quality of service. Whenever the Browser changes its location, the SpatiotemporalViewInterpreter transmits the new location to the client (e.g. to the Raster Service), whose QueryViewInterpreter makes a new query based on the updated location parameters and returns to the Browser an updated set of items (such as rasters). These updates include rasters that were not previously near enough to display, as well as rasters that were previously near enough to display but are now no longer in view.

References

- [1] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *CIKM*, pages 310–317, 2001.
- [2] B. Adamson, C. Bormann, M. Handley, and J. Macker. Negativeacknowledgment (NACK)-Oriented Reliable Multicast (NORM) Protocol. RFC 3940 (Experimental), Nov. 2004.
- [3] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.
- [4] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing Systems*, pages 262–272, 1999.
- [5] F. Bellifemine, A. Poggi, and G. Rimassi. Jade: A fipa-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents and Multi-Agents*, pages 97–108, April 1999.
- [6] M. Boulkenafed and V. Issarny. Coherency management in ad-hoc group communication.
- [7] M. Breugst, I. Busse, S. Covaci, and T. Magedanz. Grasshopper – A Mobile Agent Platform for IN Based Service Environments. In *Proceedings of IEEE IN Workshop 1998*, pages 279–290, Bordeaux, France, 1998.
- [8] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.
- [9] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.
- [10] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data, 2002.
- [11] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. pages 379–390, 2000.
- [12] K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999.
- [13] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, and T. Suel. Aqua: System and techniques for approximate query answering. Technical report, Information Sciences Research Center, Bell Laboratories, 24 February 1998. White Paper.
- [14] A. Goldberg. Information models, views, and controllers. *Dr. Dobb's Journal*, July 1990.
- [15] G. E. Grasner and S. T. Pope. A cookbook for using the model-viewcontroller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.
- [16] C. Greenhalgh and S. Benford. MASSIVE: A distributed virtual reality system incorporating spatial trading. In *International Conference on Distributed Computing Systems*, pages 27–34, 1995.
- [17] S. D. Gribble, M. Welsh, J. R. von Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Y. Zhao. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.
- [18] M. Griss. Coolagent: Intelligent digital assistants for mobile professionals - phase 1 retrospective, 2002.
- [19] F. Huet, D. Caromel, and H. Bal. A high performance java middleware with a real application, 2004.
- [20] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams, 2003.
- [21] R. Lavender and D. Kafura. A polymorphic future and first-class function type for concurrent

object oriented programming in c, 1995.

- [22] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition edition, November 1999. ISBN 0-201-31009-0.
- [23] D. Leibs and K. Rubin. Reimplementing model-view-controller. *The Smalltalk Report*, March/April 1992.
- [24] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential evaluation of continual queries. In *International Conference on Distributed Computing Systems*, pages 458–465, 1996.
- [25] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 2002.
- [26] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *The VLDB Journal*, pages 581–590, 2001.
- [27] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed agents for networking things. In *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.
- [28] M. Philippsen and M. Zenger. Java party – transparent remote objects in java. *Concurrency: Practice & Experience*, 9(11):1225–1242, November 1997.
- [29] R. Raje, H. William, and M. Boyles. An asynchronous remote method invocation (armi) mechanism for java.
- [30] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.
- [31] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the uncertain position of moving objects. *Lecture Notes in Computer Science*, 1399:310–??, 1998.
- [32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
- [33] Sun Microsystems, Palo Alto, CA. *JavaSpaces Specification*, 1.0 edition, January 25 1999.
- [34] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.
- [35] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 439–453, New York, NY, USA, 2005. ACM Press.
- [36] M. Welsh. System supporting high-performance communication and i/o in java. Master's thesis, University of California at Berkeley, October 1999.
- [37] Wikipedia. Gnutella. http://en.wikipedia.org/wiki/Gnutella.
- [38] Wikipedia. Napster. http://en.wikipedia.org/wiki/Napster.
- [39] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T-spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

4 Development with SOUSA

In this section, we illustrate the application development tasks with code from **mil.navy.nrl.cmf.sousa.Test**, a simple SOUSA program that performs updates and invokes an asynchronous RMI function. The source contains code for both a client and a server and it is more thoroughly documented than the version presented here.

Unless otherwise noted, all classes named in this section are members of the package **mil.navy.nrl.cmf.sousa.**

Developing a SOUSA Entity entails developing a main() and developing some plug-in modules. Before going into that, a brief overview of the properties and components of an Entity is in order.

An Entity has a contact port. It listens on that port at all network interfaces.

An Entity may have an authoritative State. This is where it keeps the information that it may wish to share with its clients. Clients attach ViewInterpreters to the authoritative State. The ViewInterpreters adapt the values of the State Fields into State Fields that the clients can understand. ViewInterpreters can use parameters supplied by the client in this adaptation.

An Entity has a ControlLogic. It provides the Entity with a specialized way to initiate and react to events. The events include the arrival and departure of a connection to a client, the arrival and departure of a connection to a server, and generic I/O events.

An Entity may have a multicast address and base port. Only server Entities must have them. They are used to send state change notifications to clients using a reliable multicast protocol. The first Projector that can be shared by clients will use the address mcastAddress:basePort. The next will use mcastAddress:basePort+1. Each unique shareable Projector gets its own address and port. When a Projector's last client disconnects, the Entity reclaims the port for reuse.

A server Entity may provide a number of functions to its clients via RMI. An external Object provides those functions and the functions are determined by the interfaces that the Object implements.

A client Entity may request a Receptor that supports a QoS from a server Entity. The Receptor will contain a State that has the Fields of each ViewInterpreter Class contained by the QoS. The Receptor will offer the RMI functions of the server. The RMI functions available are those of the interfaces Classes contained in the QoS and those of the interfaces implemented by the ViewInterpreters that also implement the tagging interface CommandLogic.

4.1 ***Develop main()***

Every Entity needs a main() to instantiate it, set it to running, and clean up when it terminates. Main() has the usual Java signature:

```
public static void main(String [] args)
```

Every Entity `main()` must do the following:

- Determine the communication parameters
- Create the authoritative State, giving it fields
- Create the ControlLogic
- Create the List of ViewInterpreters
- Create the List of CommandLogics
- Create the Content-Type to Renderer Map
- Create the Entity
- Optionally, schedule connections to other Entities

Listing 1 contains the main() for both the client side and the server side of the Test Entity.

```
public static void main(String[] args) {
try {
      ...
      if (args[0].equals("s")) { // Server part
            // Determine the communications parameters
            int contactPort = Integer.parseInt(args[1]);
            String mcastAddress = args[2];
            int mcastBasePort = Integer.parseInt(args[3]);

            // The server State has one field, "Foo", which
            // contains a Long.
            State authoritativeState = new State();
            authoritativeState.addField("Foo", new Long(1L));

            // Create the CommandLogic
            IntegerIncrementor commandLogic = new IntegerIncrementor();
            LinkedList commandLogicList = new LinkedList();
            commandLogicList.add(commandLogic);

            // Create the ControlLogic, ServerControlLogic,
            // and create the Entity.
            Entity e = new Entity(contactPort,
                                  mcastAddress, mcastBasePort,
                                  authoritativeState,
                                  new ServerControlLogic(),
                                  null, // Offer no ViewInterpreters
                                  commandLogicList,
                                  null /* No Renderers */);
      }
      else if (args[0].equals("c")) { // Client part
            // Determine the communications parameters
            int contactPort = Integer.parseInt(args[1]);
```

```
InetAddress remoteAddress = InetAddress.getByName(args[2]);
int remoteport = Integer.parseInt(args[3]);

// The client has no State and no multicast address.
// It's only a client.  It can't act as a server.  All
// it has is a contact port and a ClientControlLogic.
Entity e = new Entity(contactPort,
                    null /* no mcast address */,
                    -1 /* no mcast port */,
                    null /* no state */,
                    new ClientControlLogic(),
                    null /* no View Interpreters */,
                    null /* no CommandLogics */,
                    null /* no Renderers */);
// QoS using sharable session 0
QoS qos = new QoS(0);

// The server must create a Projector that has a
// TestViewInterpreter.
qos.add(TestViewInterpreter.class);

// The server must return a Receptor that implmements
// the Incrementor interface.
qos.add(Incrementor.class);

// The Entity will fetch a Receptor from the server
// when it gets time.
try {
      e.scheduleConnectTo(
            new ServerContact(remoteAddress,
                  remoteport, qos));
} catch (UnknownHostException ex) {
      System.out.println("No ServerContact! " + ex);
}
```

```
      } else {
            usage();
      }
      Thread.sleep(10000);
} catch (Exception ex) {
      System.out.println("Exception: "+ex);
      ex.printStackTrace();
}
}
```

Listing 1: mil.navy.nrl.cmf.sousa.Test.main()

4.1.1 Determine the communication parameters

Main() must determine the communications parameters for the Entity. The Entity must have a contact port. It listens on that port at all network interfaces for new connections from clients. The SOUSA runtime handles the listening and informs the Entity's ControlLogic of new connection requests.

An Entity may have a multicast address and base port. Only server Entities must have them. They are used to send state change notifications to clients using a reliable multicast protocol. The first Projector that can be shared by clients will use the address mcastAddress:basePort. The next will use mcastAddress:basePort+1. Each unique shareable Projector gets its own address and port. When a Projector's last client disconnects, the Entity reclaims the port for reuse.

Listing 1 shows main() determining all of the communications parameters from the command line arguments.

4.1.2 Create the authoritative State

An Entity may have an authoritative State. This is where it keeps the information that it may wish to share with its clients. Clients attach ViewInterpreters to the authoritative State. The ViewInterpreters adapt the values of the State Fields into State Fields that the clients can understand. ViewInterpreters can use parameters supplied by the client in this adaptation.

A State is a collection of Fields. Every Field has a String that names it and a **java.lang.Serializable** as its value. Listing 2 illustrates creating a State and adding a single Field to it. The name of the Field is “Foo” and its value is Long(1L).

```
State authoritativeState = new State();
authoritativeState.addField("Foo", new Long(1L));
```

Listing 2: Initializing the authoritative State

4.1.3 ***Create the ControlLogic***

The ControlLogic is the brain of the Entity. It reacts to and initiates events. The requirements of a ControlLogic are discussed in the section on ControlLogics, below. Main() must instantiate the ControlLogic.

4.1.4 ***Create the List of ViewInterpreters***

Every server Entity offers a (possibly empty) list of ViewInterpreters as qualities of service. This means that when a client asks for one or more of the ViewInterpreters, the Entity returns a Receptor that implements all of the interfaces implemented by each of the ViewInterpreters and that contains all of the State fields provided by each of the ViewInterpreters. All of the Field names are qualified by the fully qualified name of the ViewInterpreter class. The section on ViewInterpreters, below, explains this in more detail.

4.1.5 ***Create the List of CommandLogics***

Every server Entity offers a (possibly null) list of CommandLogics as qualities of service. A CommandLogic is a quality of service that is not implemented by a ViewInterpreter. It services method calls only. It does not contribute Fields to the State maintained in the Receptor as a ViewInterpreter does. The section on CommandLogics, below, explains this in more detail.

4.1.6 ***Create the Content-Type to Renderer Map***

Each server Entity offers a (possibly null) list of "applets" to its client. Each applet is identified by a string of the form specified for MIME Content-Types by RFC2045. The keys are Content-Type strings. The values are the Class objects of classes that implement the Renderer interface for that Content-Type. The constructors for those classes must have no arguments. When a client requests a Receptor and specifies a Content-Type in the QoS, the Entity returns the Receptor with a Renderer inside if there is an entry for the Content-Type in this Map. The keys are included in the Entity's ServerContact.

BUG in version 1.3: By RFC2045, Content-Types are case-insensitive. In this implementation, they are case-*sensitive*.

BUG in version 1.3: Entity's getServerContact method includes only one of the Content-Type keys in the ServerContact. It is not possible to predict which one.

4.1.7 Create the Entity

The Entity is alive and running the instant that its constructor returns.

4.1.8 (Optional) Schedule connections to other Entities

Once the Entity exists, it may be told to fetch Receptors from other Entities. Doing so requires three steps: creating a QoS object for some session number, adding the ViewInterpreter and CommandLogic Class objects to the QoS, and telling the Entity to schedule a connection to the other Entity at its contact address and contact port along with the QoS. Listings 3-6 illustrate the three steps.

The QoS constructor takes an integer that identifies the "session" at the server. By convention, negative session numbers indicate unsharable sessions; that is, sessions that have their own Projectors with precisely one client. Such a Projector communicates with its Receptor using a reliable point-to-point protocol. Non-negative session numbers indicate sharable sessions; that is, sessions whose Projectors may have more than one client. Such a Projector communicates with its clients using a reliable multicast protocol. The example shows the creation of a QoS for a sharable session, session 0.

```
QoS qos = new QoS(0);
```

Listing 3: Create the quality of service given the session number

Add each of the desired ViewInterpreter and ControlLogic Class objects to the QoS. In the example, qos becomes a request for the TestViewInterpreter ViewInterpreter and the Incrementor CommandLogic.

```
qos.add(TestViewInterpreter.class);
qos.add(Incrementor.class);
```

Listing 4: Add ViewInterpreters and CommandLogics to the quality of service

If the Entity desires a Renderer “applet” from the server, then set the Content-Type of the QoS. If the server can provide a Renderer for that Content-Type, then the Receptor it returns will contain one.

```
qos.setContentType(“x-sousa/x-test”);
```

Listing 5: (Optional) Add Content-Type to the QoS to request a Renderer "applet"

Finally, the Entity must schedule time to connect to the server. The scheduleConnectionTo() method requires the server's contact information in the form of listening address and listening port in addition to the QoS. When the corresponding Receptor arrives from the server, the Entity's ControlLogic is notified. See the section on ControlLogic, below, for more information.

```
try {
      e.scheduleConnectTo(
            new ServerContact(remoteAddress,
                  remoteport, qos));
} catch (UnknownHostException ex) {
      System.out.println("No ServerContact! " + ex);
}
```

Listing 6: Scheduling a connection to a server Entity

4.2 Develop the Plug-in Modules

An Entity requires up to four kinds of plug-in modules: ViewInterpreters, Service Interfaces, CommandLogics which implement the Service Interfaces, and ControlLogic.

4.2.1 Develop a ViewInterpreter that implements a quality of service

A server Entity offers information to its clients. The characteristics of this information and its delivery to clients is called *quality of service*. The implementation of a quality of service is a composition of ViewInterpreters and CommandLogics.

A ViewInterpreter is a listener. It monitors a (possibly empty) collection of Fields in an Entity's State and transforms them into a (possibly different, possibly empty) collection of Fields in response to a query or a change in the Entity's State. The transformation has two inputs: the Fields of the Entity's State and the query parameters given by the client and expressed as a State.

When a client fetches a Receptor from the Entity, it can ask for a quality of service simply by providing the corresponding Class objects of ViewInterpreters and CommandLogics in a QoS object. In the Test example, the quality of service is composed of TestViewInterpreter and Incrementor. This is reproduced in Listing 7.

```
QoS qos = new QoS(0);
qos.add(TestViewInterpreter.class);
qos.add(Incrementor.class);
```

Listing 7: Quality of service is the composition of ViewInterpreters and CommandLogics

A ViewInterpreter can act as a CommandLogic. All of the public methods of all of the interfaces that the ViewInterpreter implements are available to clients through asynchronous RMI. This is automatic. The ViewInterpreter need not implement CommandLogic.

ViewInterpreter is an abstract class. Every concrete implementation must a constructor that takes a State as its only argument. In addition, there must be implementations of `isDirty()` and `interpret()`. There may be an implementation of **`getCurrentFieldValues`**`()`. Listing 8 shows TestViewInterpreter, an implementation of ViewInterpreter from the Test example.

```
public static class TestViewInterpreter extends ViewInterpreter {
      public TestViewInterpreter(State s) {
            super(s);
            HashSet fields = new HashSet();
            fields.add("Foo");
            s.attachFieldListener(fields, this);
      }

      public boolean isDirty(State parameters){
            return true;
      }

      protected State.ChangeMessage interpret(List fieldChanges,
                                        State parameters) {
            State.ChangeMessage answer = new State.ChangeMessage();

            // Append "_BAR" to each of the Entity's Field values
            for (Iterator i= fieldChanges.iterator(); i.hasNext(); ) {
                  Field.ChangeMessage entityFCM =
(Field.ChangeMessage)i.next();
                  String barValue = entityFCM._value.toString() + "_BAR";
                  Field.ChangeMessage projectorFCM =
                        new Field.ChangeMessage(entityFCM._fname +
"_Bar", barValue);

                  answer.addFieldChangeMessage(projectorFCM);
            }
            return answer;
      }
}
```

Listing 8: A simple ViewInterpreter

4.2.1.1 *Implement the ViewInterpreter constructor*

The constructor must take a single argument, a State. The State, **s**, is the Entity's entire State. The constructor must do two things. First, it must call `super(s)`. Second, it must establish listening to the fields of interest in **s.** It does that by collecting all of their names in a `java.util.Set`, **f**, and then invoking `s.attachFieldListener(f, this)`. These requirements are illustrated in Listing 9.

```
public TestViewInterpreter(State s) {
      super(s);
      HashSet fields = new HashSet();
      fields.add("Foo");
      s.attachFieldListener(fields, this);
}
```

Listing 9: A ViewInterpreter constructor must take a State as its only argument

In the example, TestViewInterpreter listens to the Field “Foo” in the State s.

4.2.1.2 *Implement `public boolean isDirty(State parameters)`*

The SOUSA framework needs guidance about when to give a ViewInterpreter time to adapt the Entity's authoritative State for its clients. It obtains guidance from each ViewInterpreter through the `isDirty()` method. `isDirty()` returns `true` to indicate the ViewInterpreter's need to send an update to its clients. It returns `false` to indicate no need to update the clients. The implementation examines `parameters`, the query inputs from a client, and determines if their current values indicate the need to send an update. By returning `true`, the ViewInterpreter is asserting that it will have something new to transmit to the clients when `interpret()` is called.

```
public boolean isDirty(State parameters){
      return true;
}
```

Listing 10: The simplest implementation of isDirty()

Although Listing 10 illustrates a simple implementation of `isDirty()` in the Test example, an implementation that examines the Fields of interest from the Entity's authoritative State is certainly possible and desirable.

4.2.1.3 *Implement protected `State.ChangeMessage interpret(List fieldChanges, State parameters)`*

`interpret()` is the ViewInterpreter's method for adapting the Entity's State and a client's query parameters into a `State.ChangeMessage` to send back to the client. The run-time invokes sometime after a call to `isDirty()` returns `true`.

```
protected State.ChangeMessage interpret(List fieldChanges,
                                        State parameters) {
      State.ChangeMessage answer = new State.ChangeMessage();

      for (Iterator i= fieldChanges.iterator(); i.hasNext(); ) {
            Field.ChangeMessage entityFCM =
                  (Field.ChangeMessage)i.next();

            // Output field value is the Entity's field value + "_BAR"
            String barValue = entityFCM._value.toString() + "_BAR";

            // Output field name is the Entity's field name + "_Bar"
            Field.ChangeMessage projectorFCM =
                  new Field.ChangeMessage(entityFCM._fname + "_Bar",
                                          barValue);

            answer.addFieldChangeMessage(projectorFCM);
      }
      return answer;
}
```

Listing 11: interpret() adapts the Entity's State into client State

As Listing 11 reveals, `fieldChanges` is a `java.util.List` of `Field.ChangeMessage`. This implementation ignores `parameters`, which is a snapshot of the client's State.

The example adapts the Entity's State by appending "_BAR" to the value of each of the server's Fields of interest. The name of a Field in the output is its name in the server State concatenated with "_Bar".

The framework also adapts the names of the fields. The name of each field in the projected by the ViewInterpreter is the concatenation of the class name of the ViewInterpreter with an underscore and the name of the field created by `interpret()`. Names adapted in this way are called *fully qualified*. In the example, the ViewInterpreter class name is "sousa.Test$TestViewInterpreter". The Field projected by TestViewInterpreter is "Foo_Bar". Thus, the name of that Field at the client is "sousa.Test$TestViewInterpreter_Foo_Bar".

The last lines of the example's `for` loop, reproduced below in Listing 12, show how easy it is to produce the set of State changes for a client. First, create a `Field.ChangeMessage` for the changed Field. Then, insert the `Field.ChangeMessage` into the `State.ChangeMessage`.

```
Field.ChangeMessage projectorFCM =
      new Field.ChangeMessage(entityFCM._fname + "_Bar", barValue);


answer.addFieldChangeMessage(projectorFCM);
```

Listing 12: Creating a Field.ChangeMessage and adding it into a State.ChangeMessage

4.2.1.4 (Optional) Implement protected `State.ChangeMessage getCurrentFieldValues(State parameters)`

`getCurrentFieldValues()` returns a State.Change message that contains one Field.ChangeMessage for every Field of interest. It is used by the SOUSA run-time to give Fields to a new Receptor.

The Test example uses the default `getCurrentFieldValues()`, which invokes `interpret()` on a List of all of the Fields of interest.

4.2.2 Develop a Service Interface

ViewInterpreters are not the only Objects that provide quality of service. CommandLogics are also part of that equation. Each CommandLogic implements one or more Service Interfaces, where a the Service Interface corresponds to a role that an Entity is willing to assume for its clients. A Service Interface is a Java interface in which all public methods return Object. All public methods of the Service Interface are available to the Entity's clients via asynchronous RMI.

Listing 13 contains the service interface for the Incrementor quality of service.

```
public interface Incrementor {
      // Returns i + 1;
      //
      // By convention, the actual return type is Number.
      public Object increment(Number i);
}
```

Listing 13: Example service interface. Every method returns Object.

4.2.3 Implement a Service Interface with a CommandLogic

Whenever an Entity offers a Service Interface in its qualities of service, there must be a CommandLogic to implement the Service Interface. Recall from the previous section that all methods of a Service Interface must return Object. The implementations in CommandLogics may return any Serializable Object. The CommandLogic shown in Listing 14 is an implementation of the Incrementor Service Interface. Its single method, increment(), returns an Integer.

```
public static class IntegerIncrementor implements Incrementor {
      public Object increment(Number i) {
            int iVal = i.intValue();
            return new Integer(iVal + 1);
      }
}
```

Listing 14: Example CommandLogic

4.2.4 Implement a ControlLogic

The ControlLogic is the brain of the Entity. It makes changes to the authoritative State, initiates asynchronous RMI on Receptors from server Entities, and responds to events. There are a number of interesting events in the life of an Entity:

- asynchronous I/O and the completion of asynchronous RMI
- connection arrivals
- connection departures
- changes in the states of Receptors
- the passage of time

All of these are discussed in the sections that follow.

A java.lang.reflect.Proxy, which may be obtained from a Receptor, provides the apparent implementation of this interface. It is backed by a java.lang.reflect.InvocationHandler, which is the Receptor, whose invoke() dispatches calls of the interface's methods to a remote server. The InvocationHandler returns a SelectableFutureResult.

When the server returns an answer, the InvocationHandler sets the real value of the SelectableFutureResult. The type of that value is Number.

4.2.4.1 (Optional) Implement `public void handle(Selectable sel, SignalType st)`

The SOUSA run-time performs all I/O asynchronously using a mechanism like UNIX's `select()` and `fd_set. handle()` is a ControlLogic's way of learning that some I/O Object, a `Selectable`, is in a readable, writable, or error state. The important Selectable for the SOUSA application developer to know about is SelectableFutureResult. SelectableFutureResult is the Object returned by all of the asynchronous RMI calls on Receptor methods.

Listing 15 shows the default implementation of `handle(Selectable, SignalType)`, which is used by the Test example. The action for a readable Selectable (indicated by `SignalType.READ == st`) is to print the Object returned by the Selectable's read(). A

smarter implementation of handle() would write something when the Selectable is writable (`SignalType.WRITE == st`) using the Selectable's `write()` method and correct an error situation (`SignalType.ERROR == st`) after examining the Selectable's error state using `getError()`.

The client side of the Test example lets the default handle() implementation print the result of the asynchronous RMI.

```
public void handle(Selectable sel, SignalType st) {
      if (SignalType.READ == st) {
            Object obj = sel.read();
            _LOG.debug(new Strings(new Object[] {"Read ", obj}));
      } else if (SignalType.ERROR == st) {
            _LOG.error(new Strings(new Object[] {
                  sel, " threw exception ", sel.getError()}));
      } else if (SignalType.WRITE == st) {
            _LOG.error(new Strings(new Object[] {
                  sel, " is writable. WHY AM I INTERESTED IN THIS?"}));
      }
}
```

Listing 15: The default implementation of handle(Selectable, SignalType)

4.2.4.2 ***(Optional) Implement*** ***`public void projectorReadyIndication(ServerSideFSM fsm)`***

The SOUSA run-time notifies an Entity of a new connection to a client by invoking the ControlLogic's projectorReadyIndication(). The ControlLogic is now free to perform any per-client setup or other special actions. *Any class that overrides the base class projectorReadyIndication() must invoke super.projectorReadyIndication().*

4.2.4.2.1 **The ServerSideFSM fsm represents the state of the connection to the client. It has two public methods:**

- `public final ServerContact getClientContact()`

  Returns the contact information for the client: the InetAddress, port, and QoS

- `public void disconnect_Request()`

  In the current release, disconnect_Request() does nothing. In the next release, it will initiate the disconnection of the connection to the client. The SOUSA run-time will call the ControlLogic's projectorNotReadyIndication() when the connection is closed.

In the Test example, ServerControlLogic, the server's ControlLogic, uses projectorReadyIndication() to count client connections and to start a timer upon the arrival of the first client connection. This is shown in Listing 16.

```
public void projectorReadyIndication(ServerSideFSM fsm) {
      super.projectorReadyIndication(fsm);
      _clients ++;
      if (1 == _clients) { // start timer
            Clock c = getEntity().getClock();
            _alarm = c.setAlarm(1000 /* period in milliseconds */,
                  true /* recurring or not */,
                  null /* user data for AlarmHandler.handle() */,
                  this /* the Clock.AlarmHandler */);
      }
}
```

Listing 16: ServerControlLogic's projectorReadyIndication()

4.2.4.3 ***(Optional) Implement*** ***`public void projectorNotReadyIndication(ServerSideFSM fsm)`***

The SOUSA run-time notifies an Entity of the departure of a connection to a client by invoking the ControlLogic's projectorNotReadyIndication(). The ControlLogic is now free to perform any per-client cleanup or other special actions. *Any class that overrides the base class projectorNotReadyIndication() must invoke super.projectorNotReadyIndication().*

As shown in Listing 17, ServerControlLogic's projectorNotReadyIndication() stops the timer whenever there are no clients.

```
public void projectorNotReadyIndication(ServerSideFSM fsm) {
      super.projectorNotReadyIndication(fsm);
      _clients --;
      if ((_clients <= 0) && (null != _alarm)) {
            // stop timer
            _alarm.disable();
      }
}
```

Listing 17: ServerControlLogic's projectorNotReadyIndication()

4.2.4.4 ***(Optional) Implement*** ***`public void receptorReadyIndication(ClientSideFSM fsm)`***

In a client Entity, the run-time signals the arrival of a connection to a server using receptorReadyIndication(). The ControlLogic is now free to perform any per-server setup or other special actions. *Any class that overrides the base class receptorReadyIndication() must invoke super.receptorReadyIndication().*

If the ControlLogic desires to use a Renderer for outputting the state of the Receptor, receptorReadyIndication() is the place to introduce that Render to a Renderable, the object into which the Renderer renders. The Renderer can come with the Receptor as an "applet" or the ControlLogic can create it as a "plugin". The example in Listing 18 shows one way to do it.

4.2.4.4.1 **The ClientSideFSM fsm represents the state of the connection to the server. It has four public methods:**

- `public Object genProxy(Class c)`

  Returns a dynamic Proxy for the public methods of c. Listing 18 shows how ClientControlLogic invokes genProxy() to obtain an Incrementor from the ClientSideFSM.

- `public final ServerContact getServerContact()`

  Returns the contact information for the server: the InetAddress, port, and QoS.

- `public final Receptor getReceptor()`

  Returns the Receptor associated with the ClientSideFSM.

- `public final void deregister()`

Initiates the disconnection of the connection to the server. The SOUSA run-time indicates the completion of the disconnection by calling receptorNotReadyIndication().

4.2.4.4.2 The Receptor has three public methods of interest to the SOUSA application developer.

- `public State getState()`

  Returns the State of the Receptor. The field names are fully qualified as explained above in the section on the ViewInterpreter.interpret().

- `public QoS getQoS()`

  Returns the QoS associated with the Receptor.

- `public ServerContact getServerContact()`

  Returns the contact information for the server: the InetAddress, port, and QoS.

The client part of the Test example shown in Listing 18 obtains an Incrementor Proxy and invokes its `increment()` method. The return value, `answer`, is a SelectableFutureResult. If ClientControlLogic implemented `handle(Selectable, SignalType)`, it would be notified when `answer` has a value.

```
public void receptorReadyIndication(ClientSideFSM fsm) {
      super.receptorReadyIndication(fsm);

      // Introduce Renderer to Renderable.  This is optional.
      Receptor r = fsm.getReceptor();

      // “applet” Renderer provided by the server
      Renderer renderer = r.getRenderer();
      if (null == renderer) {
         // Create the Renderer “plugin”.
      }
      Renderable renderable = ...; // Create your own Renderable here
      renderer.setRenderable(renderable);
```

```
      // This is the way to obtain a Proxy that implements the
      // Incrementor interface.
      Incrementor view = (Incrementor)fsm.genProxy(Incrementor.class});
      Integer i = new Integer(32767);
      Object answer = view.increment(i);
}
```

Listing 18: ClientControlLogic's receptorReadyIndication()

4.2.4.5 (Optional) Implement `public void receptorNotReadyIndication(ClientSideFSM fsm)`

In a client Entity, the run-time signals the departure of a connection to a server using receptorNotReadyIndication(). The ControlLogic is now free to perform any per-server cleanup or other special actions. *Any class that overrides the base class receptorNotReadyIndication() must invoke super.receptorNotReadyIndication().* **This method is never invoked in Release. 1.2 or 1.3.** It will be invoked in Release 1.4.

4.2.4.6 (Optional) Implement `public boolean admitClient(ClientSideFSM.FetchRequest msg)`

Before the SOUSA run-time creates a Projector and/or a Receptor to accommodate a new client, it invokes the ControlLogic's `admitClient()` for permission. The ControlLogic gives grants permission by returning `true` and denies permission by returning `false`.

In the Test example, the server uses the default implementation, which always returns `true`.

4.2.4.7 (Optional) Implement `public void receptorStateChangeIndication(Receptor r)`

The run-time informs the ControlLogic of changes in the state of a Receptor by invoking receptorStateChangeIndication(). The ControlLogic is free to examine the Receptor State (by calling r.getState()). All Fields in the Receptor State are fully qualified as explained above in the section on the ViewInterpreter.interpret().

In the Test example, the client uses the default implementation, which does nothing.

4.2.4.8 *(Optional) Implement the passage of time.*

A common way to mark the passage of time is for the ControlLogic to implement the Clock.AlarmHandler interface. The ControlLogic then offers itself as the Clock.AlarmHandler when it adds a Clock.Alarm to the Entity's Clock. Listing 19 presents a slightly modified version of the Clock.Alarm creating code of the Test example shown in Listing 16. In the context of the Test example, `this` is the ServerControlLogic.

```
Clock c = getEntity().getClock();
Clock.Alarm _alarm = c.setAlarm(1000 /* period in milliseconds */,
                                true /* recurring or not */,
                                null /* user data for AlarmHandler.handle() */,
                                this /* the Clock.AlarmHandler */);
```

Listing 19: Creating a Clock.Alarm to mark the passage of time

The SOUSA run-time informs the Clock.AlarmHandler when the Clock.Alarm expires by calling `handle(Clock.Alarm)`.

In Listing 20, the Test example's ServerControlLogic handles the expiration of its Clock.Alarm by incrementing the value of the “Foo” field in the authoritative State by one and rescheduling the Clock.Alarm.

```
public void handle(Clock.Alarm m) {
      if (m == _alarm) {
            State authoritativeState = getEntity().getState();
            try {
                  Long oldValue =
                        (Long)authoritativeState.getField("Foo");
                  Long newValue = new Long(oldValue.longValue() + 1);
                  authoritativeState.setField("Foo", newValue);
                  m.enable();  // Reschedules the recurring Alarm
            } catch (NoSuchFieldException ex) {
                  ... // Complain
            }
      } else {
            ... // Complain
      }
}
```

Listing 20: ServerControlLogic implements the Clock.AlarmHandler interface

5 Development with IDOL

If SOUSA is the raw form of agent programming, then IDOL is the cooked form. IDOL simplifies some of the complexities and details of SOUSA programming.

To use IDOL for Entity development, the developer must write

- a ControlLogic
- an EntityIntializer that extends IdolInitializer
- a Properties file that defines, at minimum, the communications parameters and the Fields in the authoritative State and optionally defines the RFC2045 Content-Types and the corresponding Renderers the Entity offers to its clients.
- (optional) ObjectFactory classes for instantiating the authoritative State Fields from Properties

For services, main() is provided by mil.navy.nrl.cmf.sousa.idol.service.Main. For user applications, such as browsers, main() is provided by mil.navy.nrl.cmf.sousa.idol.user.Main.

5.1 Develop an EntityInitializer

IdolInitializer automatically extracts the communications parameters and the authoritative State Fields from a Properties object. IdolInitializer is an abstract class. Every concrete implementation must have a constructor that takes a Properties object as its only argument. In addition, there must be implementations of initialize_makeControlLogic() and initialize_custom().

5.1.1 Implement a constructor `public <YourNameHere>Initializer(Properties p) throws EntityInitializer.InitializationException`

The constructor uses the Properties to determine the initial values of the Entity's communications parameters and State fields. The constructor must have just one argument, a Properties Object.

The communications parameters are:

- a multicast address, a string whose value comes from the property `idol.entity.mcastAddress`
- a multicast port, an integer whose value comes from the property `idol.entity.mcastBasePort`
- a contact port, an integer whose value comes from the property `idol.entity.contactport`

In order to automatically initialize the authoritative State from the Properties, the constructor must call `super(p)` . The rules for Field naming and value initialization are defined below in the section on the Properties file. Any Fields that cannot be initialized automatically can be initialized in `initialize_custom()`.

5.1.2 Implement `protected abstract ControlLogic initialize_makeControlLogic(Properties p) throws EntityInitializer.InitializationException`

initialize_makeControlLogic instantiates the ControlLogic.

5.1.3 Implement `protected abstract void initialize_custom(Properties p) throws EntityInitializer.InitializationException`

initialize_custom may add any additional fields custom to this particular initializer and perform any other custom initializations. It may add to the IdolInitializer's list of quality of service classes using `addQoSClass(Class)`, add to the CommandLogics using `addCommandLogic(CommandLogic)`, and schedule connections to servers using `scheduleConnectTo(ServerContact)`.

5.2 Develop a Properties file

The Properties file is the key to IDOL's ease of use. It contains properties that define the communications parameters and the names and values of the authoritative State fields.

5.2.1 Communications Parameters

The communications parameters are defined by three properties:

- `idol.entity.mcastAddress` is the multicast address, a string
- `idol.entity.mcastBasePort` is the multicast port, an integer
- `idol.entity.contactport` is the contact port, an integer

5.2.2 Define idol.entity.initializer

The property `idol.entity.initializer` is the fully qualified name of a class that extends mil.navy.nrl.cmf.sousa.idol.IdolInitializer. The IDOL main() instantiates an instance of that class, then uses its public methods to obtain the values for instantiating an Entity. As an example, if the IdolInitializer mil.navy.nrl.cmf.sousa.idol.user.ClientInitializer is to be used to initialize the Entity, then the `idol.entity.initializer` property will be

```
idol.entity.initializer=\
     mil.navy.nrl.cmf.sousa.idol.user.ClientInitializer
```

5.2.3 Automatic State Initialization

The process of automatic State initialization has a few rules. The rules may appear to be complicated but in practice they are simple and easy to follow.

5.2.3.1 Define idol.initializer.attributes

IdolInitializer obtains the names of the classes that contain the names of the fields from the property `idol.initializer.attributes`, a comma separated list of fully qualified class names. IdolInitializer obtains the names of the fields by reflection on each of the classes in `idol.initializer.attributes`. The value of each public static final String field whose name ends in "FIELDNAME" becomes the name of a State field.

For example, if `idol.initializer.attributes` has the value "mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields" and QueryClientFields is defined as shown in Listing 21, then IdolInitializer finds the public static final String field POSITION_FIELDNAME and uses the value of that field, "Position", as the name of a Field in the Entity's authoritative State.

```
package mil.navy.nrl.cmf.sousa.spatiotemporal;

public final class QueryClientFields
{
     public static final String POSITION_FIELDNAME =
          "Position";
     ...
};
```

Listing 21: public static String field whose name end in "_FIELDNAME"

5.2.3.2 ***Define an initializer for each Field***

IdolInitializer looks for a factory for each Field name it discovers by reflection. The name of the factory property is the fully qualified name of the field with ".factory" appended to it. Using the example from the previous section, the fully qualified name of the Position field is `mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position`, so the name of the Position factory is `mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position.factory`.

The value of any such factory property must be an ObjectFactory. ObjectFactories implement the interface `mil.navy.nrl.cmf.sousa.util.ObjectFactory` with its single public method

```
public Object create(String prefix, Properties p)
     throws IllegalArgumentException;
```

Continuing with the example above, the Position field is an instance of `mil.navy.nrl.cmf.sousa.spatiotemporal.Vector3d` and its factory is an instance of `mil.navy.nrl.cmf.sousa.spatiotemporal.Vector3d Factory`, so the factory property for Position is

```
idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Pos
ition.factory=\
     mil.navy.nrl.cmf.sousa.spatiotemporal.Vector3dFactory
```

5.2.4 *(Optional) Define the Content-Type-to-Renderer Map*

A service Entity may define one or more mappings from RFC2045 Content-Type to Renderer class. These mappings match the Content-Type to the Renderer "applet" that the Entity makes for its clients. The Content-Type keys are included in the Entity's ServerContact. Other kinds of Entities may find different uses for these mappings.

BUG in version 1.3: By RFC2045, Content-Types are case-insensitive. In this implementation, they are case-*sensitive*.

BUG in version 1.3: Entity's getServerContact method includes only one of the Content-Type keys in the ServerContact. It is not possible to predict which one.

The Properties that define the Content-Type map all begin with `idol.initializer.contentType.` There is a `size` property that identifies the number of elements in the map. Each element is numbered and defined by one or two properties: `type` and `renderer`. The `type` property is mandatory. It is the RFC2045 Content-Type string. The `renderer` property is optional. It is the fully-qualified name of a class that implements the Renderer interface. The class must have a zero-argument constructor.

```
idol.initializer.contentType.size=3
idol.initializer.contentType.element.0.type=x-idol/x-city
idol.initializer.contentType.element.0.renderer=mil.navy.nrl.cmf.sousa.idol.us
er.CityRenderer idol.initializer.contentType.element.1.type=x-idol/x-point
idol.initializer.contentType.element.1.renderer=mil.navy.nrl.cmf.sousa.idol.us
er.PointRenderer idol.initializer.contentType.element.2.type=x-idol/x-
NoRenderer
```

5.2.5 *(Optional) Develop ObjectFactory classes*

Each kind of Field subject to automatic initialization requires an implementation of ObjectFactory to initialize it. Each such implementation must have a zero-argument constructor. The create() method makes an Object of some type given the property name prefix and the Properties. It searches the Properties for properties with names that begin with the prefix and uses their values to instantiate an Object.

Continuing with the example, Position is an instance of Vector3d and its factory is a Vector3dFactory. IdolInitializer instantiates a Vector3dFactory, f, and invokes its create() method like this:

```
Object value =
f.create("idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClient
Fields.Position", p);
```

The Vector3dFactory f expects to find properties prefix+".x", prefix+".y", and prefix+".z" in the Properties p. Clearly, in this example the prefix is "idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position " and indeed the Properties file contains precisely those three entries.

Listing 22 Shows the complete Properties file for initializing an authoritative State with a single Vector3d field named "Position".

```
idol.entity.contactport=8000
idol.entity.mcastAddress=234.23.34.20
idol.entity.mcastBasePort=4400
idol.entity.initializer=mil.navy.nrl.cmf.sousa.idol.user.ClientInitializer
idol.initializer.attributes=\
       mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields
idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position.factory=\

       mil.navy.nrl.cmf.sousa.spatiotemporal.Vector3dFactory

idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position.x=0

idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position.y=0

idol.mil.navy.nrl.cmf.sousa.spatiotemporal.QueryClientFields.Position.z=0
```

Listing 22: Example properties

6 **Under the Hood**