A Lightweight Object-Oriented Distributed Services Framework for Engineering Interactive Applications

Zachary O. Toups, Andruid Kerne, Andrew Webb
Interface Ecology Lab | Department of Computer Science and Engineering | Texas A&M University
{ zach, andruid, andrew } @ecologylab.net

ABSTRACT

We present a lightweight open source object-oriented services framework for constructing distributed applications. The congruence of internal object and externally passed messages facilitates the rapid development of correct, robust, and high-performance information-centric applications. The Object-Oriented Distributed Semantic Services (OODSS) framework is validated through its intensive use in an interaction logging service, a networked team game with hard time constraints, and a variety of other interactive systems. OODSS is lightweight, making it easy to quickly construct, debug, and deploy networked applications.

OODSS connects data to algorithms by integrating the `ecologylab.xml` information binding framework. `ecologylab.xml` efficiently translates message object instances to and from XML documents. Message declaration source code combines data fields with algorithms declared in methods. The programmer instantiates message objects. OODSS automatically transforms them to XML, sends them over the network, transforms them back into object instances on the other side, and performs remote method invocation. `ecologylab.xml` is extremely efficient, enabling high-speed network transfer. Because messages are XML, they are human-readable, simplifying debugging, yet the compression option minimizes data transfer.

Keywords
Networked applications, services, object-oriented programming, software engineering.

INTRODUCTION

We present the Object-Oriented Distributed Semantic Services (OODSS) framework for rapid development of information-centric networked applications. OODSS makes programming intricate client/server information exchange easy, without sacrificing performance. Information exchange is simplified by connecting data to algorithms by object definitions. An object class specification consists of annotated *fields* that *identify data* to be exchanged and *methods* that *define algorithms* performed on the data. Exchange is handled by using the program class to translate its instances to and from compressed XML documents, the format used for distributed message passing. This results in a framework capable of handling the needs of a distributed application with hard time constraints, but that is easy to code, understand, and debug. The framework currently supports Java clients and servers, with support for iPhone Objective C clients under development, and subsequent support for C++, C#, Python, and JavaScript planned.

Lightweight development in OODSS comes from a new architecture that layers message class definitions and an efficient [5] annotation metalanguage interpreter. `ecologylab.xml` translates message object instances to and from XML, using embedded specifications of what fields in an object are marshalled, and how, including rich expressive support for nested objects and collections.

To ensure compatibility with diverse application clients, messages are transmitted using concise, lightweight of XML. Concise messages are easy for programmers to read, and thus, debug. The flexibility of the underlying XML layer enables replicating a wide variety of existing XML dialects, such as Really Simple Syndication (RSS).

To show how OODSS functions in practice, we present real world case studies: interaction logging services, a multi-player game for teaching team coordination, GPS sensor services for building applications within Google Earth, and the combinFormation information collection visualization service. We then provide a detailed description of the layered architecture and revisit the multi-player game in depth. We conclude with discussion and future work.

CASE STUDIES

OODSS is validated by its intensive use in research and education. Applications include a networked team game, an interaction logging service, Google Earth dynamic sensor display, and the service interface to the combinFormation information composition creativity support tool. Students have used the framework on class projects.

Logging Service

OODSS forms the basis of a flexible set of logging services that assemble usage data during user studies. Applications are instrumented to produce logging operation events. These are consumed by a `Logging` listener, which abstracts log file writing. The `Logging` listener is configured to write to a local file or to send messages to a remote logging service, in which case the `Logging` object forms the application logic for a Java OODSS client. Over the past 4 years, the logging service has been used for at least half a dozen user studies, with hundreds of subjects.

Log event instances are marshalled initially, for transmission from client to server, but not subsequently unmarshalled by the service, increasing efficiency. Instead, the service writes the event objects to a file as character data so that they may be marshalled by other applications later, during log analysis and playback. Log event objects are application-specific, but logging request and response messages are general. Thus, a single logging server is automatically able to record logs from heterogeneous clients without special configuration.

Networked Team Game

Teaching Team Coordination through Location-aware Games (TTeCLoG) designs and develops digital games for teaching team skills, based on fire emergency response work practice [19]. The game requires a large amount of information be transmitted over the network in near real time, otherwise the experience of simulated presence would be lost, and the game would be unplayable. We have run the game such that each of four game clients (running on only a 1.60 GHz Intel Pentium M 725) sends the server (a 2.4 GHz Intel Core 2 Duo) a message and receives a response every 25 milliseconds; each server response is ~4.5 kilobytes long, containing over 100 XML entities. The game has been successfully played by more than 10 teams of four players. Each team played eight games, across four separate one hour sessions. A detailed walkthrough of how OODSS is used in TTeCLoG is described later; Figure 2 compares a sample TTeCLoG message with sample messages in other frameworks.

GPS Sensor Service for Google Earth

We use OODSS to develop a Keyhole Markup Language (KML) [4] service that dynamically publishes real-time GPS sensor information for interactive location tracking in Google Earth [http://earth.google.com]. This service has been used by undergraduate and graduate students to develop applications on assignments within two weeks, in a course on location aware interaction. Google Earth presents a zoomable interface to the globe with boundaries, topology, and markers. It enables publication of information on the globe through the KML dialect of XML, which specifies shapes, image overlays, and other content. Earth can be configured to read dynamic KML through specification of a URL and refresh interval. Earth sends periodic GET requests, which may optionally include information about the user's current perspective on the globe, to the KML server URL.

In one application, we run the GPS sensor services on a handheld micro-PC to publish current position and location-specific WiFi availability, for visualization with Earth, supporting research on location-based services. The server's application logic reads data from a GPS sensor to determine the user's location, and from a WiFi adapter to detect locally available networks. When Earth sends the GET request, this information is composed into KML, which Earth uses to annotate a point on the globe with WiFi availability information. A reusable set of annotated Java classes represent the KML schema. The `ecologylab.xml` translation layer translates a tree of instances of these classes into a KML document that the server publishes.

Table 1. Comparison of existing service frameworks with OODSS.

	description	advantages	disadvantages
OODSS	create distributed services by specifying Java classes	unites data and algorithms	Java-centric for now, but this will change
REST [3]	provides dynamic XML / HTML from HTTP GET / POST encoded requests	simple, widely used	linear requests lack structure, hard to understand, create; limits of HTTP GET [9]
XML-RPC [22]	allows clients to access remote data operations by sending XML-encoded calls with parameters	well-defined XML structure	complicated, deeply nested XML structure; low-level abstraction; disconnect between information and operations
SOAP [21]	as XML-RPC, but namespace qualification replaces generic XML-encoding with application-specific encoding	as XML-RPC, but more flexible, less verbose	as XML-RPC, but verbosity is somewhat reduced by moving information into schemas
RMI [14]	allows clients to run remote methods by binding interface definitions and a server link through proxy objects	abstracts network connection	little fine-grained control of serialization; Java clients/servers only
CORBA [12]	language agnostic description of network communication; use compiler to convert from definition language into programming language; use proxy objects like RMI	abstracts network connection; binds most languages	code and network communication specification defined separately
servlets [18]	Java dynamic web pages; can simulate stateful connections through cookies; provides XML / HTML	easy to create dynamic web sites	awkward to maintain state server-side; deployed through a service engine
C# web services [9]	allows server to host implementation of methods; web-methods declared using annotations; communication through SOAP	abstracts network connection	code and network communication specification defined separately
EJB services [15]	allows server to host implementation of methods; web-methods declared using annotations; communication through SOAP	abstracts network connection	requires 3 external specification files (one can be generated by offline step from annotated classes); deployed through a complex service engine

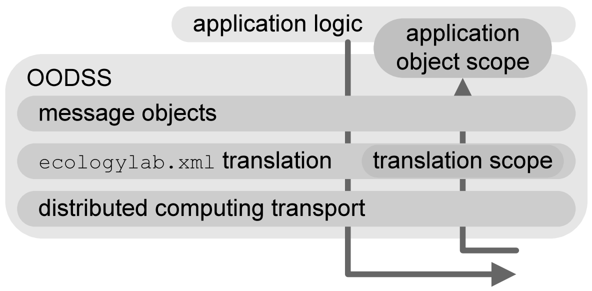

Figure 1. OODSS layers and the path of messages through them. Messages, represented as bold arrows, change form as they move through the layers.

Creativity Support Tool Collection Visualization Service

The combinFormation (cF) creativity support tool integrates searching, browsing, collecting, and thinking about information [7]. An evolving information collection is visualized in a mixed-initiative composition space: the user manipulates information directly, and invokes agents to search, compose, and crawl the web. The agents are initially directed through a set of *seeds*: search queries, documents, web sites, feeds, and curated collections. As the user expresses interest in composition elements, agents dynamically gather relevant information. cF has been used on assignments by over 400 undergraduate students per semester for seven semesters; this deployment is on-going.

cF launches as a Java Web Start (JWS) application [16] with a built-in OODSS collection visualization service running locally. Browser-based client code communicates with the service, using OODSS through a combination of HTTP POST and JavaScript.

A cF session starts with specifying seeds. This action may be performed at any time during a session. The browser, or any other client, forms an XML `set_preferences_and_seeds` message, and passes it to the cF service via OODSS. Agents operating within cF form the application logic for the visualization service. The user typically seeds cF through an HTML form, and presses the “launch” button. Browser JavaScript uses the Java Network Launching Protocol to start the JWS cF application / service. It sends ping messages to the service until it is up and running. Once cF is answering, seeding JavaScript reads the form, embeds the `set_preferences_and_seeds` XML in a hidden field in a different form, and submits it to the cF service via POST.

PRIOR FRAMEWORKS

Other frameworks, comparable in scope to OODSS, have been used to create service-based distributed interactive applications, enabling exchange of information and processing over a network. These include Representational State Transfer (REST), XML Remote Procedure Call (XML-RPC), Simple Object Access Protocol (SOAP), Java Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), servlets, C# Web Services, and Enterprise JavaBean Services (EJB). Table 1 compares their advantages and disadvantages. Figure 2 compares the representations of an example request in a multiplayer team game across some of the frameworks.

SOAP wraps service requests in namespace-qualified, verbose XML [21]. Applications specify the XML nodes understood and their semantics while the remote application specifies the applicable schema-qualified node names. The server response is in XML, structured similarly to the request. It is up to the developer to create methods that match those in the schema, which is inherently separate from the application code. In OODSS, each service’s behaviors (message class methods) are defined in the same place as their information (semantics), facilitating object-oriented software engineering and maintenance. The messages are lightweight for fast transmission and processing, and easy reading by humans.

LAYERED ARCHITECTURE

OODSS enables developers to rapidly develop federations of interoperating networked application components to create interactive applications and web services. The separation of internal and external representations of messages is eliminated. To build a distributed semantic application the programmer writes message subclass definitions, annotating fields to represent the data to be exchanged, and implements a method. Code to marshal message values and concatenate message components is not necessary. There is no complex server configuration.

The open source OODSS framework [5] is built of layers to facilitate ease of development (Figure 1). The framework provides distributed computing transport for exchanging message objects across contexts, and automatic `ecologylab.xml` translation to marshal and unmarshal these messages. The *distributed computing transport layer* uses an optimized TCP/IP connection to exchange high-level semantic messages, which are automatically encoded as XML by the `ecologylab.xml` translation layer.

Table 2. Overview of ecologylab.xml annotation metalanguage.

annotation	data type annotated	XML representation
`@xml_attribute`	scalar data type	attribute
`@xml_leaf`	scalar data type	element without children
`@xml_nested`	complex type with nested children	element with named child fields
`@xml_collection`	collection type with multiple nested children, supporting polymorphism	element with many children
`@xml_map`	map containing key-value pairs	element with many children

OODSS network message:

```
<update_client_avatar><avatar id="vbush"><pos x="10.12" y="42.42"/>…</avatar></update_client_avatar>
```

REST HTTP request:

```
http://ecologylab.net/TTeCLoGServer/updateClientAvatar?avatar_id=vbush&pos_x=10.12&pos_y=42.42…
```

XML-RPC network message:

```
<methodCall><methodName>ClientAvatarUpdate.performService</methodName>
  <params><param><value><struct>
    <member>
      <name>avatar</name>
      <value><struct>
        <member>
          <name>id</name>
          <value><string>vbush</string></value>
        </member>
        <member>
          <name>pos</name>
          <value><struct>
            <member>
              <name>x</name>
              <value><double>10.12</double></value>
            </member>
            <member>
              <name>y</name>
              <value><double>42.42</double></value>
            </member>
          </struct></value>…
    </struct></value></param></params>
</methodCall>
```

SOAP network message (`tteclog` namespace for TTeCLoG Service)**:**

```
<soap:Envelope xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema" xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <tteclog:clientAvatarUpdate xmlns:tteclog="http://ecologylab.net/tteclogns">
      <tteclog:avatar xsi:type="tteclog:ClientAvatar">
        <tteclog:id xsi:type="xsd:string">vbush</tteclog:id>
        <tteclog:pos xsi:type="tteclog:Vector2d">
          <tteclog:x xsi:type="xsd:double">10.12</tteclog:x>
          <tteclog:y xsi:type="xsd:double">42.42</t:y>
        </tteclog:pos>…
      </tteclog:avatar>
    </tteclog:clientAvatarUpdate>
  </soap:Body>
</soap:Envelope>
```

Figure 2. Sample remote service call messages to the Teaching Team Coordination through Location-aware Games (TTeCLoG) service in OODSS, REST, XML-RPC, and SOAP network communication protocols. The OODSS message is lightweight and clear, with semantics defined through high-level Java object declarations and, thus, not included in the message itself. High-level Java message declarations are described in top of Figure 5.

Because the REST message is linear, we must represent structures by qualifying variable names. XML-RPC can transmit nested `struct`s (sets of key-value pairs), consequently the XML-RPC message is structured, but long and redundant; the structures around the data are larger than the data itself by an order of magnitude. In the SOAP message, verbosity is reduced by relying on a set of XML-Schema namespaces, however, these add overhead. (Function calls are italicized; variable names, values in bold.)

The `ecologylab.xml` *translation layer* unmarshals incoming XML messages to form OODSS message object instances and vice-versa. Metalanguage annotations succinctly describe how data structures are marshalled and unmarshalled [8]. These annotations are efficiently interpreted at runtime without a pre-processor [5], facilitating rapid development while maintaining computational efficiency.

Message objects unite data and algorithms to define services. The programmer authors message object classes with metalanguage (Table 2) to simultaneously define what data is transported, how it is represented internally, and how it is represented when serialized. Service algorithms that operate on the data are defined by implementing methods in message object class declarations that are automatically invoked when a message is received. When a

Java OODSS client is used, responses are handled similarly. Non-Java OODSS clients develop their own flows of control for processing server responses.

A set of message class declarations that define a service are represented formally as a *translation scope*. This scope succinctly specifies which `ecologylab.xml` annotated classes will be used for translating incoming XML messages. The classes' metalanguage annotations define how incoming XML is automatically translated into a tree of object instances. The server can then automatically execute the methods specified in the message object.

In a client, *application logic*, specified by the programmer, sends and receives OODSS message class instances and manipulates state in the *application object scope*, a set of state objects. On the server, application logic can be implemented through concurrent threads of control that also respond to and manipulate state that is likewise connected to message objects through the application object scope.

The rest of this section describes the Object-Oriented Distributed Semantic Services architecture from the developer's perspective. Programmers use the `ecologylab.xml` annotation metalanguage to specify the strongly typed data of message classes. They implement the algorithms that act upon the data by overriding a method. Services are defined by grouping message class declarations into translation scopes, which can be composed, supporting component-based design. Service-wide and client-specific object scopes facilitate development by providing context. Distributed computing performance is optimized to support building mission-critical interactive applications.

Data: Annotation Metalanguage

Software engineering is facilitated by simplifying the way programmers specify the information semantics that allow transmission of arbitrarily structured data over a network. With the open source `ecologylab.xml` binding framework [5], developers create Java classes, in which specified data fields map to XML structure, defining how a Java object is marshalled to XML and later unmarshalled [8]. Further, such classes can be authored from XML specifications or samples to "capture" XML from the wild, enabling interoperability with existing standards and platforms, such as Google Earth KML and RSS.

Semantics are described through field declarations, annotated with the concise `ecologylab.xml` metalanguage (Table 2), to define exactly what scalar fields and nested objects will be serialized into XML messages of any complexity, and how [8]. Because the annotated fields in the subclass represent information elements in the XML message, the class functions as a template for the XML message sent over the wire. *The message translation structure is defined where it is most easily manipulated by the programmer: in the programming language.*

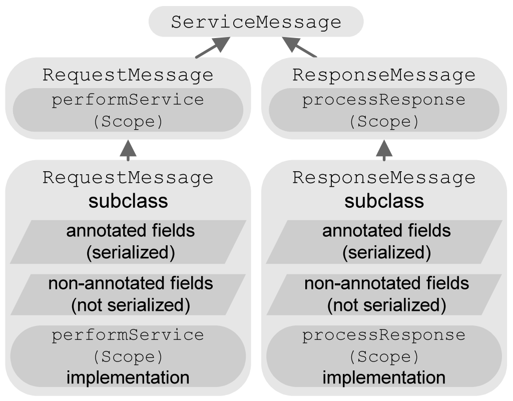
ServiceMessage
RequestMessage
performService (Scope)
ResponseMessage
processResponse (Scope)
RequestMessage subclass
annotated fields (serialized)
non-annotated fields (not serialized)
performService (Scope) implementation
ResponseMessage subclass
annotated fields (serialized)
non-annotated fields (not serialized)
processResponse (Scope) implementation

Figure 3. Inheritance diagram of message object classes.

Metalanguage directs `ecologylab.xml` in translating between Java objects and XML, resulting in a tight binding between strongly typed objects and their serial form. Fine-grained specification of what fields are marshalled and how is embedded directly in the code, uniting internal and serialized data structures. Table 2 presents an overview of the metalanguage. More detail can be found in [8].

The metalanguage and translation layer enable concise custom XML messages, facilitating debugging and conserving system resources, such as heap space and network bandwidth. Heap thrashing is a significant constraint in large-scale applications' performance [1]. Network bandwidth load affects responsiveness in distributed interactive applications, such as multiplayer games [13] and groupware. Figure 2 shows an example of the XML representations of a request in a multiplayer game service, using different messaging frameworks. The OODSS representation is lightweight and human-readable, in sharp contrast to those of SOAP and XML-RPC.

`ecologylab.xml` translates class names and field names into XML elements and attributes [8]. Camel-case conversion is automatically performed by default. For example, `UpdateClientAvatar` in Java becomes `update_client_avatar` in XML (Figure 2). The root element tag of any XML document is bound to a class name. Subsequent tags are bound to field names. Options are provided for binding collection elements, including support for polymorphism.

Algorithms: Override Method Declaration

To develop a service, the programmer authors message object classes that define a service by implementing certain method signatures from the base class and declaring appropriate data fields, connecting algorithms with data. Two base classes serve as the foundation for specifying algorithms: `RequestMessage` and `ResponseMessage`, each of which subclass `ServiceMessage` (Figure 3).

Request/Response - `ServiceMessage` *Subclasses*

`ServiceMessage` subclasses embody a service; they are the core mechanism by which a developer creates distributed computation. Instances of these classes are marshalled using `ecologylab.xml`, and sent over the network. At the destination, they are unmarshalled and their methods are invoked upon remote state objects automatically.

To create a service, the developer extends the `RequestMessage` class, implementing a method with the `performService(Scope)` signature, and declaring data fields with metalanguage for transmission to the server. Once this class is stored in the server's translation scope, a client developer can instantiate the `RequestMessage` subclass, and send marshalled instances to the server for processing.

On the server, OODSS uses a translation scope to identify the matching `RequestMessage` subclass, instantiates the appropriate object(s) using the XML data, and invokes the `performService(Scope)` method, using the `Scope` of state variables to change the state of the server and/or retrieve information from it. The outcome of the method invocation is an instance of a `ResponseMessage` subclass, which carries information back to the client. In a *Java* OODSS client, the `processResponse(Scope)` method of the `ResponseMessage` subclass will automatically be invoked in the same way.

OODSS servers are interoperable with clients other than *Java* OODSS clients. For *web browser* OODSS clients, there are three mechanisms for communicating with the server. The first two use JavaScript to submit a request as an HTML form object using the POST method. In one case, the XML is inserted as a value in a form field (as in combinFormation launch). In the other, the browser's `XMLHttpRequest` JavaScript object is used with `content-type="text/xml"`. The third method embeds the XML in the URL (with URL encoding), resulting in an HTTP GET request. These mechanisms are supported by OODSS natively. Alternatively, a server can be configured to automatically respond to HTTP requests with a pre-determined `ResponseMessage`, as is used for the Google Earth GPS service.

Unlike CORBA [12] and SOAP [21], which require that programmers declare interfaces externally in IDL or Schema, in OODSS, messages take the form of annotated program objects that specify semantics directly, promoting object-oriented software engineering.

A private API can be maintained while providing public services. Message class definitions need not be identical for both server and client, as long as the class definitions "agree" on an XML representation. This enables, for example, the specification of message base classes used by the client, and then directly extended by translation-compatible instances on the server, integrating implementation-specific service code.

Defining and Composing Services

A service is defined by a set of enhanced message and constituent class declarations bound together by a translation scope, defining the set of messages a server can translate. In `ecologylab.xml`, a lexical translation scope efficiently provides a set of bindings between XML element tags and metalanguage-enhanced classes [8]. An OODSS server is configured to provide one or more services by instantiating with a translation scope of `ServiceMessage` subclasses, and a port number. Java OODSS clients also use translation scopes, in order to translate XML responses from the server into message objects. The case study (Figure 5) shows the flow of messages from XML to `ServiceMessage` subclass instances and back again.

A single translation scope is typically used to bind together the message classes that correspond to an application component. To promote component-based software engineering, translation scopes can be composed with multiple inheritance, enabling the formal composition of service message types. For example, OODSS includes a translation scope for basic servers (with messages such as those for initiating a connection), from which one for authenticating servers inherits (adding messages for user authentication). These scopes are extended further with application-specific messages, for example, for TTeCLoG, GPS Sensor Services for Google Earth, combinFormation, and interaction logging.

A translation scope is specified using a name and an array of `Class` literals, from which `TranslationScope` automatically extracts XML tags for translation [8]. Alternative tags can be provided for a class, enabling maintenance of backward compatibility when developers need to change the XML specification for an application. Each `TranslationScope` is cached, by name, in a global name space from which a static `get(…)` accessor method will retrieve them after its first invocation, maximizing efficiency. Our practice is to create a static class that encapsulates a set of translations for a component, with such a `get` accessor.

Providing Context for Servicing Clients

For many applications (such as multiplayer games and Web 2.0 clients), it is useful to maintain context between service requests, so OODSS provides sessions which persist between messages and across unexpected disconnects. Access to and modification of information on the server is accomplished through an *application object scope*, which provides global read-only access to state variables, and a lexically chained *client session scope*, which serves to insulate clients from one another while providing a space for them to store information.

Session

A *session* is a series of transactions between a single client and a server. Sessions, and the `Scope` that maintains context across message invocations within each, persist between messages and across unexpected disconnects. Invoking the `connect(…)` method on a Java OODSS client automatically establishes a session (either restoring one in progress or starting new) with the server by sending an `InitConnectionRequest` message, unless the application has been customized to do otherwise (HTTP-based services do not require an initialization request, it is not a part of the protocol). The session connection is stateful, persisting until it is closed, unless, for example, the client is a web browser, which will close the connection automatically. Sessions are identified by a unique token. By using the token, a client that has been disconnected can reconnect, even from another IP (supporting mobile applications, e.g., in case the WiFi access point shifts), automatically restoring its previous session.

Application Object Scope

An application object scope provides references to state data for messages coming into the OODSS server (and the Java client; see Figure 4). Although the server's application object scope may not be modified directly, a message object may access and modify the scope's referenced objects through their interfaces. Scopes store hashed sets of `String` to `Object` bindings, which a message object may retrieve and then access inside its methods (`performService` or `processResponse`). `ServiceMessage`s document the bindings their services require; the developer populates the application object scope with the required bindings for the message object types in its translation scope and passes it to the server constructor.

Application object scopes provide flexibility: different applications may share the same messages classes, but bind different `Object`s (which implement the same `interface`) to the same `String`, based on need. For example, one could have several servers that create visualizations of message content in different ways, but each could use the same message code.

Client Session Scope

To provide context for a client's session, separate from that of other sessions, a client session scope is instantiated for each new connection by lexically chaining the server's application object scope. The resulting scope is read/write for client messages, while the chained application object scope is read-only. This scope persists context across messages for a single client session. When a session is finished, the client session scope is finalized.

The `performService` and `processResponse` methods of `ServiceMessage` subclasses access and operate on objects from the session scope. These methods may store context data in the session scope, while accessing data from the chained application scope. This provides global context through which clients may communicate with the server application, but are isolated from each other's data.

Application Logic

To drive sending messages from the client, and perform auxiliary computation on state objects in scopes on both the client and the server, application logic is provided by the programmer. In its simplest form, the application logic instantiates message classes, and invokes the `sendMessage(RequestMessage)` method on a Java OODSS client. Complex applications may use multiple threads to read and modify the contents of the application object scope on either the client or the server, such as resolving game state changes by opponents on a multi-player game server.

DISTRIBUTED COMPUTING OPTIMIZATIONS

OODSS uses Java New I/O (NIO) [17] to optimize performance. NIO supports multiplexed network I/O and bulk data buffer transfers using efficient native operating system calls when available. The OODSS server and Java client implementations are dual-threaded, with one thread devoted to network I/O and another to executing incoming messages, allowing network hardware to gather data concurrently with message processing.

To enable HTTP compatibility and improve performance, OODSS utilizes HTTP-like headers. For HTTP servers, messages may include any of the standard HTTP headers [20]. The `content-length` and `content-type` character encoding headers are always used, maximizing flexibility and compatibility.

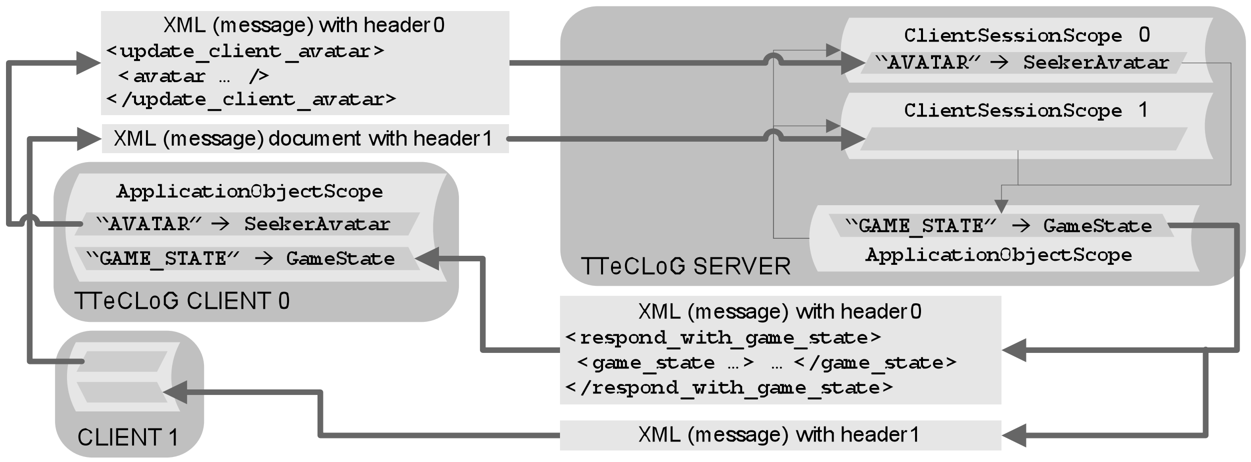

Figure 4. A set of TTeCLoG clients share object references for their `SeekerAvatar` instances with the server, while the server shares its `GameState`. State is updated by sending message class instances (`update_client_avatar`) to the server. The client receives a `respond_with_game_state` response message. Light arrows indicate a reference to an object from another object; heavy arrows indicate the flow of information through the system.

DETAILED CASE STUDY: MULTIPLAYER TEAM GAME

Our validation of OODSS is based on using it to construct high-performance applications. In this section, we present a simplified walkthrough of a real world case study from one of these applications. In this section, we briefly introduce the game design, followed by an in-depth look at its OODSS implementation. Key components of OODSS are described: the `ServiceMessage` subclasses that define message data, structure, and behavior together; translation scopes to define available services; object scopes that enable safe distributed operation; and sessions between clients and the game server.

TTeCLoG Game Design

The TTeCLoG game is designed to teach teamwork skills, based on fire emergency response work practice. To support a team structure similar to that of firefighting, game players take on one of two roles: seeker or coordinator. Seekers search for and collect hidden goals in the real world, while being tracked using GPS with wireless micro-PCs. Seekers' coordinates determine their avatar's location in a virtual world overlay, which contains invisible threats, controlled by a remote game server. To assist the seekers, the coordinator uses an overhead map of the virtual world, and communicates with the seekers using a radio. Seekers share information with each other and the coordinator in order to succeed.

The stationary game uses distributed laptop computers; seekers move their avatars using the keyboard. The OODSS components built for the stationary game will be re-used for a mixed reality version, in which play takes place in the real world, replacing the seekers' keyboards with GPS.

The shared virtual world is maintained by a server that runs the game as its application logic (moving threats, etc.) and uses OODSS to share the current state of the virtual world with each of the clients (both seeker and coordinator). Each seeker's client updates the state of its avatar in the virtual world, indicating the actions taken by the player.

OODSS Implementation

The TTeCLoG game uses OODSS to update and share the game state. Each client reports the current state of the player's avatar to the server, while the server responds to each client with the most recent state of the entire game (Figure 4).

On a seeker's client, the current state of that player's avatar within the virtual world is represented by a `SeekerAvatar` class instance that tracks Cartesian position and other state variables (such as health). `SeekerAvatar` is annotated with `ecologylab.xml` metalanguage, defining its XML representation, enabling its state to be stored or transmitted over the network (Figure 5). The `SeekerAvatar` instance processes GPS readings / keyboard input, computes the avatar's location, and sets the avatar's state from the game server's state.

The server stores a collection of `SeekerAvatar` instances, one for each client, as a part of a `GameState` object. `GameState` stores the shared virtual world overlay and the location and status of each of the seekers. When a client joins the game, a `SeekerAvatar` is instantiated with a starting location. This information is passed to the client, which will maintain the `SeekerAvatar`'s state from then on. Figure 4 shows how `SeekerAvatar` instances are updated between clients and the server. Each client sends `ClientAvatarUpdate` messages over the network; these are incorporated into the master `GameState` instance.

Service Walkthrough

In this section, we describe a typical message exchange between the server and client, following Figure 5. We assume that server and client have already initialized their connections.

In TTeCLoG, the client consists of application logic and a Java OODSS client. The application logic uses state variables from the application object scope to resolve player action in the game.

A. Sending a request to the server. The client application logic initiates a service request by invoking `sendMessage(RequestMessage)`. In TTeCLoG, the message is `UpdateAvatarRequest`, sent at a rate of 40Hz. This `RequestMessage` subclass instance is automatically translated into XML using `ecologylab.xml` and sent to the server.

B. Server receives request message from client. The message consists of an HTTP-like header that includes a `content-length` line, followed by XML describing the `UpdateClientAvatar` instance.

C. Translating the message. The message is translated by `ecologylab.xml`, using the translation scope. For TTeCLoG, the translation scope includes the `UpdateClientAvatar` message and the `SeekerAvatar` state object, among others, allowing a game client to update its `SeekerAvatar` instance in the server's game state. Figure 5 includes a sample of some of the class definitions used by the translation scope, including data members and annotations.

The `translateFromXML` method looks up the appropriate class in the translation scope using the tag name from the XML (in this case, the `update_client_avatar` XML tag maps to the `UpdateClientAvatar` class). The result of translation is an instance of `UpdateClientAvatar`, which contains the current state of the `SeekerAvatar` instance from the client. The `UpdateClientAvatar` instance's fields are populated with the data in the XML message.

D. Performing the service. The `UpdateClientAvatar` class overrides `RequestMessage`'s `performService(Scope)` method. The client session scope contains the server's copy of the client's `SeekerAvatar`. The client session scope is lexically chained from the application object scope, which provides read-only access to the server's `GameState` object.

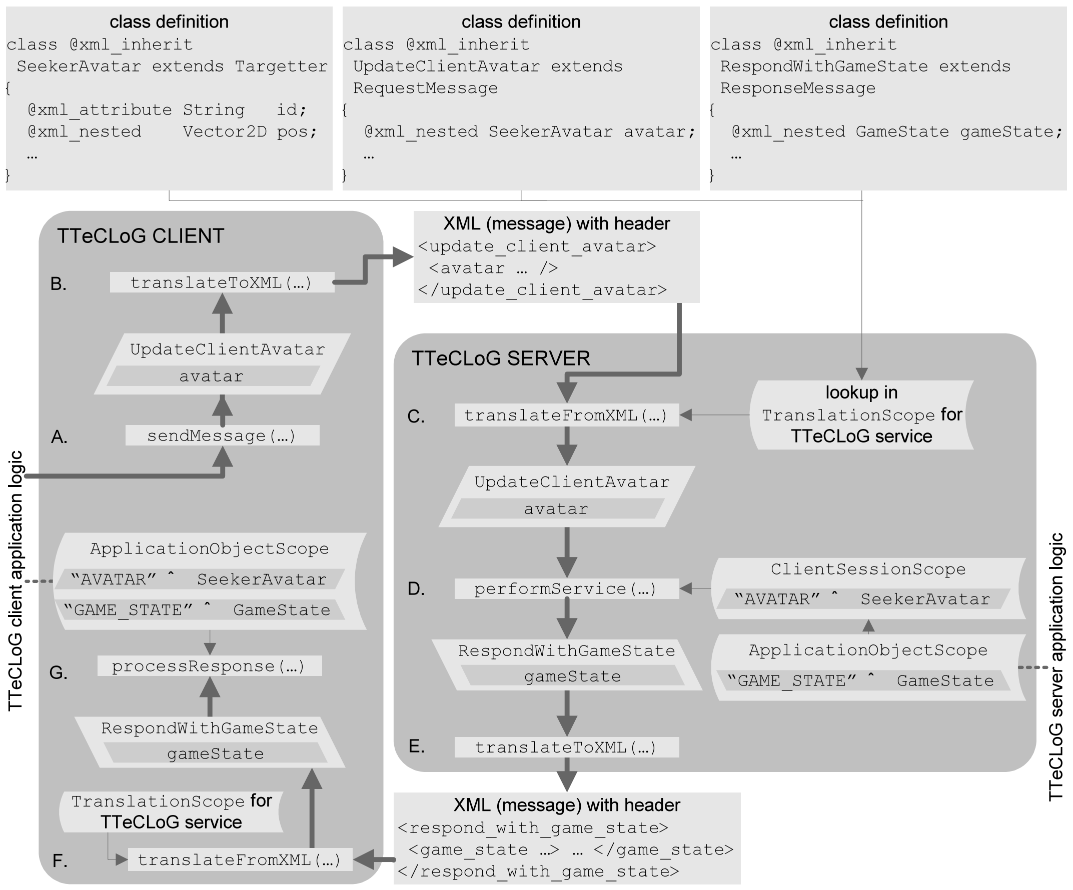
class definition
```
class @xml_inherit
 SeekerAvatar extends Targetter
{
  @xml_attribute String   id;
  @xml_nested    Vector2D pos;
  …
}
```
class definition
```
class @xml_inherit
 UpdateClientAvatar extends
 RequestMessage
{
  @xml_nested SeekerAvatar avatar;
  …
}
```
class definition
```
class @xml_inherit
 RespondWithGameState extends
 ResponseMessage
{
  @xml_nested GameState gameState;
  …
}
```
XML (message) with header
```
<update_client_avatar>
 <avatar … />
</update_client_avatar>
```
XML (message) with header
```
<respond_with_game_state>
 <game_state …> … </game_state>
</respond_with_game_state>
```

Figure 5. Life cycle of a message through the TTeCLoG client and server using OODSS. The game client application logic sends a message to the server, starting at A. and moving clockwise to G. On the server, class definitions in the translation scope enable translation of the incoming XML message from the client into a Java object. The object, a `UpdateClientAvatar` instance, supplies the player's state (in the form of a `SeekerAvatar` instance) to the server. The `performService(Scope)` method incorporates the update into the `GameState` object, then sends a `RespondWithGameState` instance, that carries the `GameState` back to the client via the same mechanism.

`performService` updates the server's version of the `SeekerAvatar` to reflect the state sent by the player's client application.

The last step in the `performService` method returns an instance of a subclass `ResponseMessage`, `RespondWithGameState`, which is sent back to the client. It contains a copy of the game state stored on the server, retrieved from the application object scope.

E. Translating the response. The server automatically marshals the `RespondWithGameState` instance, prepends an HTTP-like header, and sends it back to the client over the network.

F. Client receives response message from server. Like the server, the OODSS client receives XML with HTTP-like headers and translates them into `ResponseMessage` subclass instances. The incoming messages, which will be received in response to a prior `RequestMessage`, are translated into subclass instances of `ResponseMessage`, using the TTeCLoG translation scope. The result is the instance of `RespondWithGameState` that was sent by the server.

G. Processing the response. Once a `ResponseMessage` instance has been received, its `processResponse` method is automatically invoked on the client's application object scope. The client's copy of the shared virtual world (`GameState` instance) is updated to reflect the server's state.

Sessions and Initialization

When a player joins the game (TTeCLoG client connects to the server), the TTeCLoG application client initializes the connection, starting a session. The session is started by a short handshake sequence: a TCP/IP connection, an `InitConnectionRequest` from the client, and an `InitConnectionResponse` from the server. This results in a shared session identifier specified by the server. Once the session has started, it can be resumed later if the network fails (for example, if a wearable computer loses its network connection). A `SeekerAvatar` instance is initialized (as above). A reference to it is passed to the server's game state and added to the client session scope, chained from the server's application object scope. By sharing the reference to the client's avatar, the client updates its state in the client session scope, but is unable to change the avatars of other players. The server is able to perform game state computations based on the avatars.

CONCLUSIONS AND FUTURE WORK

OODSS facilitates rapid development of succinct and correct networked interactive applications. Efficiency and fast response are retained, while application development is simplified. The framework supports object-oriented integration of internal and distributed message data structures and service specification through message overriding and translation scopes. Programmers author request and response messages as field declarations with metalanguage annotations embedded in the same Java classes that implement application logic. Ease-of-use results from the automatic marshalling of complex values, obviating the need to write code that assembles messages. Further, because message semantics are defined in the application language, they can be manipulated with tools such as Eclipse [2]. The interaction logging service supports instrumenting interactive applications and collecting user-study from clients across the network. Demanding distributed applications such as networked games are supported with clear semantics, facilitating rapid development.

OODSS is lightweight first in that the programmer writes less code to build a distributed application. Representing service message semantics in the application programming language, instead of in an external schema as with SOAP, or in the value declarations of XML-RPC, results next in lightweight XML messages, which use less heap space for encoding and decoding and less human cognition during debugging. Further, the use of XML enables interoperability with existing systems, enabling developers to build on established standards.

OODSS develops a novel integration of the `ecologylab.xml` translation layer with network transport and object-oriented software engineering. Instead of schemas, *class definitions* specify the *semantics* of information exchange, *data structures* optimized for internal computation, and *operations* in one place. Translation scopes, which can be composed, bind together the message declarations that correspond to software components. The set of services supported by a server is defined through the set of message subclasses present in its environment, as specified in its operative translation scope. The result is to reinvigorate object-oriented software engineering's original mantra of integrated structure that promotes *software clarity* and *maintenance*.

Having developed a robust implementation in one cross-platform programming language, future work will develop support for other popular programming languages. We will also investigate developing OODSS extensions that provide interoperability with XML-RPC and SOAP, so that existing technologies can be easily incorporated.

REFERENCES

1. Christ, R., Halter, S.L., Lynne, K., Meizer, S., Munroe, S.J., Pasch, M. SanFrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal 39*, 1 (2000), 4-20.
2. Eclipse Foundation. Eclipse. http://www.eclipse.org.
3. Fielding, R.T. Representational State Transfer (REST). *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. Diss., U. California, Irvine, 2000.
4. Google. KML Documentation. http://code.google.com/apis/kml/documentation.
5. Interface Ecology Lab. ecologylab.xml. http://ecologylab.net/xml.
6. Interface Ecology Lab, OODSS, http://ecologylab.net/oodss.
7. Kerne, A., Koh, E., Smith, S.M., Webb, A., Dworaczyk, B. combinFormation: Mixed-initiative composition of image and text surrogates promotes information discovery. *ACM Trans. Information Systems 27*, 1 (2008), 1-45.
8. Kerne, A., Toups, Z.O., Dworaczyk, B., Khandelwal, M. A concise XML binding framework facilitates practical object-oriented document engineering. *Proc. ACM Document Engineering* (2008), 62-65.
9. Microsoft. Article 208427: Maximum URL length is 2,083 characters in Internet Explorer. http://support.microsoft.com/kb/208427.
10. Microsoft. Creating and Accessing Web Services (Visual C#). http://msdn.microsoft.com/en-us/library/ms228289(VS.80).aspx.
11. Nielsen, J. *Usability Engineering*. Morgan Kaufmann, 1994.
12. Object Management Group. CORBA: Core Specification. http://www.omg.org/docs/04-03-12.pdf.
13. Pellegrino, J.D., Dovrolis, C. Bandwidth requirement and state consistency in three multiplayer game architectures, *Proc. ACM NetGames* (2003), 52-59.
14. Sun. Java Remote Method Invocation. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp.
15. Sun. Java Web Services At a Glance. http://java.sun.com/webservices/.
16. Sun. Java Web Start Technology. http://java.sun.com/products/javawebstart/.
17. Sun. JDK 5.0 New I/O-related APIs & Developer Guides. http://java.sun.com/j2se/1.5.0/docs/guide/nio/.
18. Sun. JSR-154 Java Servlet 2.4 Specification. http://jcp.org/en/jsr/detail?id=154.
19. Toups, Z.O., Kerne, A. Implicit coordination in firefighting practice: Design implications for teaching fire emergency responders. *Proc. ACM Computer Human Interaction* (2007), 707-716.
20. W3C. Hypertext Transfer Protocol – HTTP/1.1. ftp://ftp.isi.edu/in-notes/rfc2616.txt.
21. W3C. SOAP specifications. http://w3.org/TR/soap.
22. Winer, D. XML-RPC Specification. http:/xmlrpc.com/spec.