A software architectural design method for large-scale distributed information systems

To cite this article: Hassan Gomaa et al 1996 Distrib. Syst. Engng. 3 162

View the article online for updates and enhancements.

Related content
- Configuring scientific applications in a heterogeneous distributed system
  Patrick T Homer and Richard D Schlichting
- Client-server programs analysis in the EPOCA environment
  Susanna Donatelli, Nicola Mazzocca and Stefano Russo
- Workflow management systems on top of OSF DCE and OMG CORBA
  Alexander Schill and Christian Mittasch

Recent citations
- Mara Nikolaidou and Nancy Alexopoulou
- M. Nikolaidou et al
- M. Nikolaidou et al
A software architectural design method for large-scale distributed information systems

Hassan Gomaa†, Daniel Menascé‡∥ and Larry Kerschberg§+

† Department of Information and Software Systems Engineering, ‡ Department of Computer Science and ∥ Department of Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, 22030-4444, USA

Received 12 September 1995

Abstract. This paper describes a software architectural design method for large-scale distributed information systems. The method, which is part of an integrated design and performance evaluation method, addresses the design of client/server software architectures, where the servers need to cooperate with each other to service client requests. The goal of this software architecture is to provide a concurrent message based design that is highly configurable. The method is illustrated by applying it to the design of a complex software system, the Earth Observing System Data and Information System (EOSDIS) Core System.

1. Introduction

A large-scale distributed information system (LDIS) is characterized by having a large number of users, a diverse user population, diversity in user requirements, high data intensity, diversity in data types stored, and function distribution (Menasce et al 1995). This paper describes a software architectural design method for large-scale distributed information systems and gives an example of its use. The method is part of an integrated design and performance evaluation method for large-scale distributed information systems described in Menasce et al (1995).

The method is oriented to the design of client/server software architectures, where the servers need to cooperate with each other to service client requests. The goal of this software architecture is to provide a concurrent message based design that is highly configurable. The concept is that the software architecture can be mapped to many different system configurations.

The method is illustrated by applying it to the design of a complex software system, the Earth Observing System Data and Information System (EOSDIS) Core System. EOS is a NASA programme mission to study the planet Earth. EOSDIS is a large-scale geographically distributed system that will handle large volumes of data sent from several satellites orbiting the Earth. The authors were part of an interdisciplinary team composed of earth scientists, computer and information scientists developing an independent architecture for the the EOSDIS Core System (ECS).

Section 2 of this paper presents an overview of the integrated design and performance evaluation method. Section 3 describes the software architectural design method for large-scale distributed information systems. Section 4 presents an overview of the Earth Observing System Data and Information System. Section 5 describes the application of the design method to the EOSDIS Core System.

2. The integrated design and performance evaluation method

2.1. Overview

Performance models play a crucial role in the design of complex information systems. They are useful to distinguish among a variety of alternatives, both good and bad, assess the impact of architectural choices, predict potential bottlenecks, size hardware components, and evaluate if a proposed architecture will meet the performance requirements under the expected workload.

This section describes an integrated design and performance evaluation method called the performance oriented system design method (figure 1). The main thrust of this method is to ensure, by successive refinements, that the architecture meets performance goals set forth in the requirements analysis and specification phase. The method is iterative and the dashed lines in figure 1 indicate feedback loops.

There are five basic inputs to the method (shown as clouds in figure 1): functional requirements, user model, performance requirements, system cost and technology constraints, and system component performance.
characteristics. The functional requirements specify the functions to be performed by the system. The performance requirements specify the requirements on performance when executing any of these functions (e.g. maximum response time values, minimum throughputs). The user model describes the typical interactions between users and the system. The user model also provides quantitative information on the frequency with which users interact with the system, as well as the resource requirements per interaction (e.g. an earth scientist studying ocean circulation models will typically browse twenty 3 MByte images and then will run an ocean circulation model that requires an average of 500 MFLOPs). System cost and technology constraints indicate the cost constraints on the system as well as what type of networking, operating system, hardware and software technologies constrain the design space. Finally, the system component performance characteristics specify the performance related features of the components to be used in the system design. Examples are processor speeds, I/O subsystem bandwidth, and network speeds.

2.2. Domain modelling

A domain model is developed to reflect the interaction among the main system components in order to satisfy the functional requirements. A domain model is a problem-oriented specification for the application domain, which captures the similarities and variations of the family of systems (Parnas 1979, Batory and O’Malley 1992) that compose the domain. Given a domain model of an application domain, an individual target system (one of the members of the family) is created by tailoring the domain model given the requirements of the individual system. In a domain model, an application domain is represented by means of multiple views, such that each view presents a different aspect of the domain (Gomaa 1993a, 1995). This work was based on the NASA-sponsored investigation into the Evolutionary Domain Life Cycle Model (Gomaa et al 1992a, b) and the associated prototype software, the Knowledge-Based Software Engineering Environment (KBSEE) (Bosch et al 1995, Gomaa et al 1996).

2.3. Design of client/server software architecture

The resulting domain model is then used to derive a client/server software architecture, which depicts the message exchanges between clients and servers in the system. The design method used in this step addresses the design of large scale distributed information systems, which execute on geographically distributed nodes supported by a local or wide area network. Typical applications include distributed client/server systems, where the servers need to cooperate with each other. The design method is described in more detail in section 3.

2.4. System architecture specification

In this step of the method, the type, number of components of each type, and the connectivity used to link them together is specified. The number and type of components selected is constrained by the system cost and technology constraints. In the initial design, this is the only guideline for the system architecture. In successive iterations of the method, the results of the performance assessment step are taken into account to help refine the system architecture.

2.5. Creation of event sequence scenarios

Event sequence scenarios are created by taking a user model, which provides a detailed description of the user interactions with the system (viewed as a black box), and
mapping them to the client/server software architecture. An event sequence scenario, which is similar to a use case (Jacobson et al. 1992), describes the sequence of events resulting from the interaction among application objects in the client/server architecture, responding to the external inputs described in a user model. The events are numbered on an event sequence diagram and are described in the accompanying text.

2.6. Performance annotation of event sequence scenarios
The event sequence scenarios are further annotated with performance parameters such as request arrival rates, data volumes per request, server processing and I/O requirements per request.

2.7. Software/hardware mapping
The client/server software architecture drives a first-cut at the system architecture. The client/server software architecture and the system architecture are used to generate a software/hardware mapping that associates logical servers to physical elements such as processors and network segments. The components of the system architecture are assigned performance characteristics (e.g. network segment speeds, router latencies, I/O subsystem bandwidth, processor speeds). Then, the performance annotated scenarios, the software/hardware map, and the system architecture performance characteristics are combined to generate input parameters to a performance model.

2.8. Performance modelling
The performance model is based on analytical methods to solve mixed (i.e. open/closed) queuing networks (Menascé et al. 1994). The outputs of the performance model include response times and throughputs for each type of request submitted to the system. An analysis of the results of the performance model reveals the possible bottlenecks. If the architecture does not meet the performance objectives, architectural changes at the hardware and/or software level have to take place. These changes, guided by the outputs of the performance model, may be to the system architecture, software/hardware mapping, or domain model specification. Successive iterations ensure that the final design meets the performance objectives.

3. Design of client/server software architecture

3.1. Introduction
The software architectural design method for large-scale distributed information systems is an extension of the CODARTS (concurrent design approach for real-time systems) design method (Gomaa 1993b) called the CODARTS/DIS (CODARTS for distributed information systems) design method. Typical applications include distributed client/server systems, where the servers need to cooperate with each other. With CODARTS/DIS, a large-scale distributed information system (LDIS) is structured into subsystems. A subsystem is defined as a collection of concurrent tasks executing on one physical node. However, more than one subsystem may execute on the same physical node. Each physical node consists of one or more interconnected processors with shared memory.

The goal of this software architecture is to provide a concurrent message based design that is highly configurable. The concept is that the software architecture can be mapped to many different system configurations within the framework of the client/server system architecture. Thus, for example, the same application could be configured to have each subsystem allocated to its own separate physical node, or have all or some of its subsystems allocated to the same physical node. To achieve this flexibility, it is necessary to design the application in such a way that the decision about mapping subsystems to physical nodes does not need to be made at design time, but is made later at system configuration time. Consequently, it is necessary to restrict communication between tasks in separate subsystems to message communication. It is assumed that a distributed kernel provides a transparent message communication capability between concurrent tasks.

There are three main steps in designing a large-scale distributed information system consisting of subsystems that can be configured to execute on distributed physical nodes:

(a) Distributed information system decomposition. Structure the distributed information system into subsystems that potentially could execute on separate nodes in a distributed environment. As subsystems can reside on separate nodes, all communication between subsystems is restricted to message communication. The interfaces between subsystems are defined. A set of subsystem structuring criteria is used for determining the client and server subsystems.

(b) Subsystem decomposition. Structure subsystems into concurrent tasks and information hiding modules. Since by definition, a subsystem can only execute on one physical node, each subsystem can be designed using a design method for non-distributed concurrent systems, such as DARTS, ADARTS or CODARTS (Gomaa 1993b). Thus, tasks within the same subsystem, which by definition always reside on the same physical node, may use inter-task communication and synchronization mechanisms that rely on shared memory.

(c) Distributed information system configuration. Once a distributed information system has been designed, instances of it may be defined and configured. During this stage, the subsystem instances of the information system are defined, interconnected, and mapped onto a hardware configuration (Magee et al. 1994).

In this paper, emphasis is on the distributed information system decomposition, which includes defining the message interfaces between subsystems, and distributed information system configuration.

3.2. Distributed information system decomposition
In this phase, the LDIS is decomposed into subsystems. Subsystem structuring criteria are provided to help the designer structure the system into subsystems that satisfy the performance objectives. The software architectural design method for large-scale distributed information systems is an extension of the CODARTS (concurrent design approach for real-time systems) design method (Gomaa 1993b) called the CODARTS/DIS (CODARTS for distributed information systems) design method. Typical applications include distributed client/server systems, where the servers need to cooperate with each other. With CODARTS/DIS, a large-scale distributed information system (LDIS) is structured into subsystems. A subsystem is defined as a collection of concurrent tasks executing on one physical node. However, more than one subsystem may execute on the same physical node. Each physical node consists of one or more interconnected processors with shared memory.

The goal of this software architecture is to provide a concurrent message based design that is highly configurable. The concept is that the software architecture can be mapped to many different system configurations within the framework of the client/server system architecture. Thus, for example, the same application could be configured to have each subsystem allocated to its own separate physical node, or have all or some of its subsystems allocated to the same physical node. To achieve this flexibility, it is necessary to design the application in such a way that the decision about mapping subsystems to physical nodes does not need to be made at design time, but is made later at system configuration time. Consequently, it is necessary to restrict communication between tasks in separate subsystems to message communication. It is assumed that a distributed kernel provides a transparent message communication capability between concurrent tasks.

There are three main steps in designing a large-scale distributed information system consisting of subsystems that can be configured to execute on distributed physical nodes:

(a) Distributed information system decomposition. Structure the distributed information system into subsystems that potentially could execute on separate nodes in a distributed environment. As subsystems can reside on separate nodes, all communication between subsystems is restricted to message communication. The interfaces between subsystems are defined. A set of subsystem structuring criteria is used for determining the client and server subsystems.

(b) Subsystem decomposition. Structure subsystems into concurrent tasks and information hiding modules. Since by definition, a subsystem can only execute on one physical node, each subsystem can be designed using a design method for non-distributed concurrent systems, such as DARTS, ADARTS or CODARTS (Gomaa 1993b). Thus, tasks within the same subsystem, which by definition always reside on the same physical node, may use inter-task communication and synchronization mechanisms that rely on shared memory.

(c) Distributed information system configuration. Once a distributed information system has been designed, instances of it may be defined and configured. During this stage, the subsystem instances of the information system are defined, interconnected, and mapped onto a hardware configuration (Magee et al. 1994).

In this paper, emphasis is on the distributed information system decomposition, which includes defining the message interfaces between subsystems, and distributed information system configuration.
decompose the system into subsystems that can execute on distributed nodes. For a given application, there may be several instances of a given subsystem. Since the goal is to have a distributed cooperating client/server architecture, client and server subsystem structuring criteria are provided as follows.

### 3.2.1. Client subsystem structuring criteria.

(a) **Real-time control.** This client subsystem controls a given aspect of the system. The subsystem receives its inputs from the external environment and generates outputs to the external environment, usually without any human intervention.

(b) **Data collection.** This client subsystem collects data from the external environment. In some cases, it may analyse and reduce the data before sending it on for storage. An example of a data collection subsystem is a sensor subsystem that collects raw sensor data from a variety of sensors, converts the data to engineering units, before sending them to a server for storage.

(c) **User services.** This client subsystem provides the user interface and a set of services required by a group of users. There may be more than one user services subsystem, one for each class of user. For example, in a factory automation system, there may be one user services subsystem for factory operators and a different user services subsystem for factory supervisors. A user services subsystem interacts with one or more server subsystems to obtain the data, which it then displays to the user.

### 3.2.2. Server subsystem structuring criteria.

Although client/server applications often refer to file servers or database servers, it is necessary in a LDIS to distinguish between several types of server, including:

(a) **Archival server.** This server subsystem handles the storage of all types of data. This type of server may be further specialized into archival servers of different types.

(b) **Metadata server.** This server subsystem manages the collection of metadata relative to the data managed by the archival server.

(c) **Processing server.** This server subsystem handles processing requests to process data received from a data collection client or another processing server.

(d) **Query server.** This server subsystem manages the processing of both ad-hoc and predefined queries.

(e) **Scheduling server.** This server subsystem schedules processing requests, allocating requests to the available processing servers. Some LDIS applications have a set of scheduling servers, which collectively implement a global scheduler.

(f) **Configuration management server.** This server subsystem monitors the operating conditions of a LDIS node, collects statistics about the utilization of its various resources, and reconfigures the node when necessary to cope with failures and performance degradation.

(g) **Catalogue management server.** This server subsystem maintains a directory of all objects managed by the LDIS. The collection of all catalogue managers collectively maintain a global directory of LDIS objects. The catalogue managers are used to locate LDIS objects.

(h) **User management server.** This server subsystem maintains information about registered users, their profiles, accounting, and security information.

(i) **Server coordination.** This server subsystem coordinates the activities of other server subsystems. For example, the server coordination subsystem may maintain a directory to indicate which server subsystem is best suited to respond to a query.

### 3.2.3. Designing subsystem interfaces.

As subsystems potentially reside on different nodes, all communication between subsystems is restricted to message communication. Tasks in different subsystems communicate with each other by means of loosely coupled message communication or tightly coupled message communication (figure 2).

Loosely coupled message communication is either by means of FIFO message queues or priority message queues. In distributed environments, loosely coupled message communication is used wherever possible for greater flexibility. Group communication, where the same message is sent from a source task to all destination tasks who are members of the group (referred to as multicast communication) is also supported.

Tightly coupled message communication is either in the form of single client/server communication or multiple client/server communication. In both cases a client sends a message to the server and waits for a response; in the latter case a queue may build up at the server. In a client/server architecture, it is also possible for a server to delegate the processing of a client’s request to another server, which then responds directly to the original client.

### 3.3. Distributed information system configuration

During information system configuration, a given information system is instantiated and mapped to a distributed configuration consisting of multiple physical nodes connected by a network. During this phase, decisions have to be made about what subsystem instances are required since some subsystems can have more than one instance, how the subsystem instances should be interconnected, and how the subsystem instances should be allocated to nodes.

During information system configuration, the following activities need to be performed:

(a) **Component instantiation.** Instances of the information system components are defined. For each subsystem type, where more than one instance can exist in an information system, it is necessary to define the instances desired. For those subsystems that are parametrized, the parameters for each instance need to be defined. Examples of subsystem parameters are sensor names, sensor limits, and alarm names.

(b) **Interconnection of subsystem instances.** The information system architecture defines how subsystems communicate with one another. At this stage, the subsystem instances are connected together, such that one component’s output port is connected to another component’s input port (Magee et al 1994).

(c) **Mapping to hardware configuration.** The subsystem instances are mapped to physical nodes. For example, two
subsystems could be configured such that they each could run on a separate node or alternatively they could both run on the same node.

4. Earth observing system data and information system

The method is illustrated by applying it to the design of a complex software system, the Earth Observing System (EOS) Data and Information System (EOSDIS) Core System. EOS is a NASA program mission to study the planet Earth. A series of satellites with scientific instruments aboard will be launched starting in 1997. These satellites will collect data about the atmosphere, land and oceans. An estimated one terabyte of raw data will be sent to the Earth every day. Raw data coming from the NASA satellites is first received at the White Sands complex in West Virginia. After some initial level of calibration, it is sent for archival and further processing at a collection of geographically distributed centers called distributed active archive centers (DAACs).

The raw data received by the DAACs is called level 0 data. Level 0 data is used to generate level 1 data, defined as reconstructed, unprocessed instrument data at full resolution, time-referenced, and annotated with ancillary information. Environmental variables at the same resolution and location as the level 1 data are derived to generate level 2 data. A set of variables mapped onto uniform spacetime grid scales, with some consistency and completeness, are called level 3 data. Finally, the output from scientific models or results from analyses of lower level data is referred to as level 4 data.

5. Example of software architecture for large-scale distributed information system

The software architecture for the EOSDIS Core System (ECS) concentrates on the science data processing segment, which includes the satellite data acquisition, processing and storage, as well as the scientific users accessing EOS archival data. For this exploratory architecture study, the emphasis has been on analysing the aspects that are common to the family of systems that compose the domain model, in other words the kernel of the domain model. In addition, because of the highly distributed nature of EOSDIS, the ECS domain model concentrates on the object communication view, which is a highly concurrent and distributed view of the system, in which the concurrent objects communicate with each other by means of messages (Gomaa 1993a, 1995). Another important view is the object model (Rumbaugh et al. 1991), which considers the data modelling perspective, and is described in Kerschberg et al. (1996).

5.1. Domain model of ECS

The domain model focuses on the client/server nature of the system. It consists of the client information management subsystem (IMS), representing the client based access by the users, which interacts with the distributed active archive center (DAAC), representing the server functionality provided by the system. The distributed active archive center (DAAC) is an important concept in the problem domain and it is modelled as a collection of cooperating servers. As there are multiple DAACs, there are multiple collections of these servers. In addition, as there are multiple clients, there are multiple instances of the client IMS.
As the functionality of the domain model is carried through to the client/server architecture, this paper focuses on the latter.

5.2. Client/server software architecture

5.2.1. Overview. The overall distributed design of the client/server software architecture is shown in figure 3. The design consists of distributed subsystems communicating by means of messages. Each subsystem consists of one or more concurrent objects, which are implemented as concurrent tasks (processes).

The client information management subsystem (client IMS), shown in figure 3, is an example of a client based user services subsystem, as it executes on the user node and interacts directly with the user. It provides local processing and caching of EOS data. It also maintains a directory of services provided by ECS. The are several instances of a client IMS, one for each user.

The client IMS processes the original query from the user. It looks up its local directory to determine where the data or service requested by the user resides. It decomposes the global query into DAAC specific queries and sends out the queries to each DAAC, where they are received by the server information management subsystem.

The client IMS communicates with the different DAAC subsystems and also other non-EOS servers, such as the NOAA, the National Oceanic and Atmospheric Administration, server. In figure 3, two instances of the DAAC subsystem, DAAC (n) and DAAC (m), are shown although there would typically be several. Each DAAC is configured to have the appropriate instances of the server information management subsystem (server IMS), product generation subsystem (PGS), and data archiving and distribution subsystem (DADS).

The PGS is a processing server subsystem which processes incoming satellite data and sends them to the DADS, which is an archival server subsystem that stores the various data products. The server IMS is a server coordination subsystem, which coordinates user access to the archived data products.

A client IMS determines from its local directory the DAAC location of the data product requested by the user and sends a product request to the server IMS on the appropriate DAAC. The server IMS looks up its directory to determine which archival server maintains the data product requested by the user and forwards the request to that server. The message from the server IMS contains the identifier of the original requesting client IMS, so that the responding server can send the data product directly to the client.

Figure 3 also shows how DAACs can communicate with each other. Thus a PGS can request data from another DAAC by sending a product request message to the server IMS for that DAAC. The server IMS forwards the message to the appropriate product server, which responds by sending the data product to the requesting PGS. The PGS subsystem is decomposed further as shown in figure 4 while the DADS subsystem is decomposed further as shown in figure 5.

5.2.2. Design of product generation subsystem. Figure 4 shows the concurrent subsystem architecture of the product generation subsystem (PGS), which processes level 0 (L0 in figure 4) data obtained from the EOS satellites, to generate the level 1–4 data products, which are then archived in the DADS. Browse data are generated for those who wish a less detailed view of the data. In addition, metadata are generated, providing information about the data and algorithms used to derive a given archival product. EOS data products are derived from raw EOS scientific data, using instrument calibration data, and applying scientific algorithms. At a given level n, a level n archival product is generated as well as level n metadata and optionally level n browse data. In figure 4, these are collectively shown as Ln products.

The product scheduling server, an example of a scheduling server, receives product generation requests from multiple sources. It adds these requests to a priority ordered product generation list. It sends processing requests and reprocessing requests to the level n processing servers, which execute the product generation algorithms. It also sends product requests to other DAACs for products that are needed as input to the various processing stages on this DAAC, shown as requested Ln product in figure 4.

5.2.3. Design of data archive and distribution subsystem. The data archiving and distribution subsystem (DADS) deals with the archiving of the standard data products received from the PGS, as well as the distribution of the standard products, and processing of requests for standard products originating from scientific users. Standard products include level 1–4 products, metadata products and browse data products.

A generic design of the data archive and distribution subsystem is given in figure 5. It consists of several instances of the product server subsystem, duly tailored and instantiatted as needed for a given DADS. There is also an offline staging server, which receives products that have been requested for offline delivery.

The product server subsystem receives product and metadata requests from the server IMS. It also receives products from the PGS for archival. It sends metadata to the server IMS and products to users, either online via the appropriate client IMS, or offline through the offline staging server.

5.2.4. Design of product server subsystem. The design of a product server subsystem is shown in figure 6. In order to provide improved throughput, the product server subsystem is designed as a concurrent server subsystem. The design is generic and has to be tailored for any specific server.

Each of the services provided by the concurrent server subsystem is implemented as a concurrent task type, which is instantiated to provide the service. There is also a product archive manager (PAM) task which receives client requests. The PAM instantiates a server task to handle each client request, e.g. a reader task to handle a read request. The reader task reads the product archive and either:
Figure 3. EOS client/server software architecture diagram.

(a) sends the requested product directly to the user via the client IMS, if this was an online request, or
(b) sends the requested product to the offline staging server if this was an offline request.

After servicing the client request, the reader task terminates.

A user may also subscribe to a product in which case the subscription request is forwarded to the subscription manager task, which updates the subscription list. The request is for the client to be either registered or removed from the subscription list. Product updates from the PGS are received by the writer task. The writer task updates the product archive and then sends an update notification message to the production distribution task. The production distribution task then reads the subscription list and sends a copy of the new data to each registered client, as indicated in their subscription specification. This is handled by multicast message communication. However the time at which the data are sent is dependent on the priority, type of request, and requested delivery time of each subscription request.

Since there are multiple readers and writers accessing the product archive, an appropriate synchronization algorithm must be used, such as the mutual exclusion algorithm or the multiple readers/multiple writers algorithm. In the latter case, multiple readers are allowed to access the product archive concurrently; however a writer must have mutually exclusive access to the archive.
5.2.5. Product distribution  The various instances of the client IMS send user requests for products to the appropriate server IMS at the various DAACs. For metadata and browse data product requests, the server IMS services the requests on arrival. However, for archival product requests, the server IMS first estimates the amount of data to be delivered by querying the metadata server and then, if the data volume is above a certain threshold, sends a schedule to the user. For small amounts of data, the request is forwarded to the archival server directly.

The server IMS queries the metadata for the given product to determine the size and frequency of the requested data granule. For volumes of data above the threshold, it sends a schedule to the user via the client IMS indicating the estimated amount of data requested, an estimated schedule of delivery, with options for online or offline delivery.

5.3. Distributed information system configuration

The client/server software architecture is intended to be highly configurable. The design of several of the server subsystems is generic, so they have to be configured for a specific server subsystem instance. As an example, consider the configuration of the product server subsystem:

(a) Specialization. The product server subsystem can be adapted for a specific need through specialization, e.g. to create a level 1 archival server, a level 3 browse data server, or a level 2 metadata server.

(b) Instantiation. Certain parameters can be passed to a given instance of the product server subsystem at instantiation time.

(c) Component interconnection. The actual binding of concurrent components to each other is done. For example, the level 1 product processing server is bound to the following instances of the product server subsystem: level 1 archival server, the level 1 metadata server, and the level 1 browse data server.

(d) Mapping to hardware configuration. Each component is mapped to a specific hardware node.
5.4. Event sequence scenarios

As described in section 2.5, in order to validate the functionality of the client/server software architecture, several event sequence scenarios were created reflecting earth science user interactions with the architecture. The black-box user interactions with the system are described in user models. The event sequencing scenarios were developed by applying the earth science user models to the client/server software architecture, thereby identifying how scientific data products are generated, archived, and accessed by users. There are two main types of scenarios:

(a) Push scenarios, which show the generation of level 1–4 data products and their archival in the DAACs.
(b) Pull scenarios, which show earth scientists making requests for archived data products.

Several scenarios were developed, one of which is described here. This oceanographic scenario is the ocean heat transport and storage scenario developed by F. Webster and J. Churgin. This scenario involves accessing archived data products from various EOS instruments carried on EOS satellites. These data products are derived from the data collected by the MIMR (Multifrequency Imaging Microwave Radiometer) and MODIS (Moderate-Resolution Imaging Spectroradiometer) instruments. Data are also obtained from NOAA, the National Oceanic and Atmospheric Administration.

The event sequence scenario showing user interaction with the architecture is depicted graphically on an event sequence diagram, as shown in figure 7. Figure 7 shows the oceanographic scenario in which a client IMS interacts with servers at two DAACs. Each DAAC has a server IMS and various archival servers. The event sequence is described below with the event sequence numbers corresponding to figure 7.

(1–2) The earth science user requests the client IMS to display information on sea surface temperature (SST) from the local directory. The local directory provides information on various oceanographic products available from NASA satellites including two items listed as SST (level 2), one from MIMR and the other from MODIS, and one listed as gridded SST. These are all displayed in a window at the user’s workstation.

(3–12) The user requests metadata on the MIMR SST and MODIS SST. The local directory manager determines that the MIMR metadata is stored at the NASA Marshall Space Flight Center DAAC and the MODIS metadata is stored at the NASA Goddard Space Flight Center DAAC. It sends queries to the server IMS at Marshall and the server IMS at Goddard. The queries are handled identically at both locations. Consider the Marshall server IMS, which
looks up its directory to determine where the information is stored. It sends a metadata product request to the level 2 metadata server passing on the identifier of the originating client. The level 2 metadata server receives the request and reads the metadata archive. The level 2 metadata server sends the requested metadata directly to the client IMS. The metadata is displayed at the user’s terminal in the MIMR metadata window.

(13–22) The user makes product requests for MIMR and MODIS browse data. This is handled in a similar way to metadata.

(23–26) The user queries the directory and requests data from NOAA.

(27–37) For a more detailed analysis, the researchers decide to use the level 2 MODIS data, and send a request for delivery of the MODIS data product. The local directory manager determines that the product is archived at NASA Goddard and sends a query to the server IMS at Goddard. The Goddard server IMS looks up the directory to determine where the metadata for this product is stored. The server IMS then sends a query to the level 2 metadata server requesting information on the file size and frequency of data collection. The metadata server sends the metadata to the server IMS, which makes an estimate of how much data the user’s request would entail and when it could be delivered. The server IMS sends the proposed schedule to the client IMS with product delivery options. The user selects online delivery of the product. The client IMS sends the product request to the server IMS. The server IMS sends the level 2 product request to the level 2 archival server, passing on the id of the originating client. The request is added to the queue of requests for this server with the schedule of when the product is scheduled for delivery to user. At the scheduled time, the level 2 archival server creates a reader task to read the metadata archive. The level 2 archival server sends the requested product directly to the client IMS. The product data is displayed at the user’s terminal in the MODIS product data window. The user invokes a local algorithm to process the data.
H Gomaa et al

6. Conclusions

This paper has described a software architectural design method for large-scale distributed information systems, which is part of an integrated design and performance evaluation method. The goal of this software architecture is to provide a concurrent message based client/server software architecture that is highly configurable. The concept is that the software architecture can be mapped to many different system configurations.

The paper has also described the application of the method to a case study of a large-scale distributed information system, the Earth Observing System Data and Information System Core System. EOSDIS is a large-scale geographically distributed system that will handle large volumes of data sent from several satellites orbiting the Earth.

Unlike many software design methods which are initially developed by applying them to toy problems, this design method was developed iteratively by applying it to the design of a complex system. Thus many of the client and server subsystem structuring criteria were determined by finding specific needs for them in the EOSDIS core system. The client/server software architecture was also developed iteratively and its viability was demonstrated by creating detailed earth science user scenarios that were then applied to the architecture in order to identify how scientific data products are generated, archived, and accessed by users.

Acknowledgments

This work was partially supported by Hughes Applied Information Systems. This work built on the NASA-sponsored investigation into the Evolutionary Domain Life Cycle Model and the associated prototype software, the Knowledge-Based Software Engineering Environment. The authors would like to acknowledge the many useful discussions they had with the ECS Independent Architecture Study Group at GMU led by Menas Kafatos. In particular, they would like to thank Jim Churgin, Ferris Webster, Berrien Moore III, and Jim Kinter, for explaining to them the different aspects of earth science and user scientific requirements for EOSDIS. Special acknowledgments are due to Jim Churgin and Ferris Webster, who developed the Ocean Heat Transport and Storage Scenario, which forms the basis of the event sequence scenario described in section 5.4. They would also like to thank Frank Carr for his considerable assistance in constructing the domain model and client/server software architecture.

References


Gomaa H 1993a A reuse-oriented approach to structuring and configuring distributed applications Software Eng. J. March 61–71


——1995 Reusable software requirements and architectures for families of systems J. Syst. Software March


——1992b A Knowledge-based approach to domain modeling: application to NASA’s payload operations control centers J. Telematics Informatics 9 (3/4)


Jacobson I, Christenson M, Jonsson P and Overgaard G 1992 Object-Oriented Software Engineering (Reading, MA: Addison-Wesley)

Kerschberg L, Gomaa H, Menascé D and Yoon J P 1996 Data and information architectures for large-scale distributed data intensive information systems Proc. 8th Int. Conf. on Scientific and Statistical Database Management (Stockholm, 1996)


