Context Model Sharing in the FIPER Environment

Zhao, Shuo
GE Corporate Research and Development Center, Niskayuna, NY 12309

Sobolewski, Michael
GE Corporate Research and Development Center, Niskayuna, NY 12309

Abstract

The goal of the FIPER environment [1] is to form a federation of distributed services that provide engineering applications and tools on a network. A highly flexible software architecture has been developed, in which engineering tools like computer-aided design (CAD), computer-aided engineering (CAE), product data management (PDM), optimization, cost modeling, etc., act as distributed service providers and service requestors. Service providers can enter the federation by registering with a service broker and publish the services through a process of discovery and join. The individual services communicate via so-called context models, which are abstractions of the master model of a particular product. The context model represents a business object, which has the ability to retain its state and integrity while being concurrently accessed. Therefore, a data persistence paradigm has been developed to ensure persistence across the network. The framework for sharing context models in the FIPER environment consists of a presentation layer, along with a thin web client, business logic layer and repository layer. It is implemented using Java, remote method invocation (RMI), Jini™, JavaSpaces, and servlet.

1. Introduction

The development of complex mechanical systems such as turbine engines is a highly coupled multidisciplinary process. In a market with ever increasing demands in terms of life cycle cost, environmental aspects (noise, emissions, and fuel consumption), and performance, the availability of accurate analytical tools during the design process is a given and ceases to be a discriminator between the various competitors. It is, therefore, the application of these tools and their automated interaction in a robust computational environment, which may decide over success or failure of a specific project through reduction of design cycle time and avoidance of costly rework because of availability of high-fidelity information earlier in the design process. At the same time, especially in a multi-national company, design increasingly takes place at spatially distributed locations, potentially all over the world, where all participants in the design process need constant real-time access to all relevant up-to-date product information. In light of these challenges, GE has teamed with Engineous Software, BFGoodrich, Parker Hannifin, Ohio Aerospace Institute, and Ohio and Stanford Universities in a four-year effort to develop a "Federated Intelligent Product EnviRonment" (FIPER) [1] under the sponsorship of the National Institute for Standards and Technology-Advanced Technology Program (NIST-ATPTM). FIPER strives to drastically reduce design cycle time, and time-to-market by intelligently automating elements of the design process in a linked, associative environment, thereby providing true concurrency between design and manufacturing. This will enable distributed design of robust and optimized products within an advanced integrated web-based environment.

The FIPER architecture is simultaneously web-centric, service-centric and network-centric. This architecture houses a pool of federated services. The web-centricity enables transparent web-based access to the globally distributed data and the pool of services. The individual services can act within this framework, both in the role of providing services (server mode) and requesting services (client mode). When requesting services, the FIPER infrastructure also brokers the requests, delegating them to the appropriate registered service.

Data persistence control is a critical element of the FIPER environment, in which data from its context models and distributed business logic are available globally. In this model, the distributed clients might request services concurrently for the same data. Without data persistence control, data integrity would be compromised because clients could use common data concurrently: clients could modify the same data at the same time. Data persistence can also provide database independence, so that multiple database vendors can be adopted.

A context model is the basic element of FIPER data structures; it forms the essential structure of the data being processed. A context model in the FIPER system is represented as a tree-like structure of context nodes. A data node is where the actual data resides. The context denotes an application domain, and a context model is its context with data nodes as leaf nodes appended to its context paths. A partial context model structured for a turbine airfoil mechanical analysis is depicted in Figure 1.

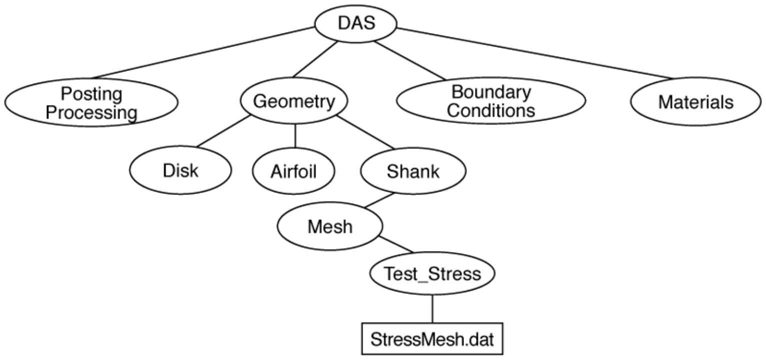

Figure 1: Example context model for turbine analysis showing the tree-like node structure

Ovals in the figure represent context nodes, which in turn form the paths of this model. The rectangle shape indicates the data node, which is located at the leaf.

The framework for the data persistence service is shown in Figure 2. Since the FIPER environment is a federation of distributed services, the ability to share and access context models is itself one of the FIPER services. The first step for clients sharing a context model in a persistent and concurrent manner is finding the service provider, as they would for any other services. The details for locating a service provider are presented later.

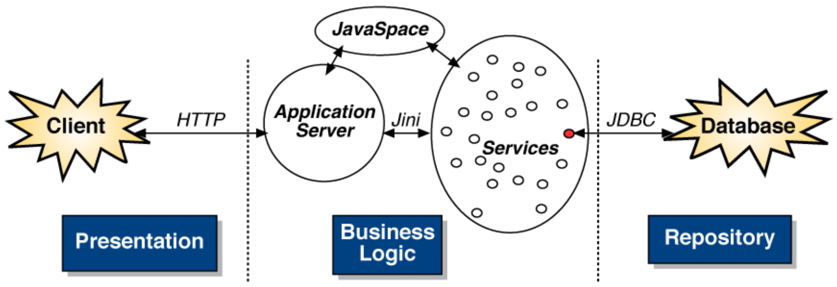

Figure 2: The three-tier architecture for FIPER context sharing allows full separation of data storage from

The three centricities of the FIPER system are clearly identified in Figure 2. The client accesses the application server through an HTTP portal that indicates the web-centric approach. The network-centric concept is shown (middle) by the network representation (large oval) that contains various service providers (small ovals). Since all the providers form a network-centric environment, the application server discovers and communicates with a particular provider via its catalog of services or drops a task into a JavaSpace. Within the network, all FIPER services are connected to multiple spaces and execute relevant tasks by taking a task and returning results to the space. An example of the service-centric concept is shown by the small highlighted circle at the right of the larger oval, which represents the data persistence service. The persistence service uses JDBC (Java Database Connectity) to talk with the underlying database (shown on the right). As far as the client is concerned from the service-centric perspective, getting a service from the desired provider is the ultimate goal.

The three-tiered FIPER approach creates a layered software architecture in which there is complete separation of the data storage (Repository Layer) and the service-based business logic (Business Logic Layer), from the web user interfaces (Presentation Layer). In this architectural model, the business rules are mapped into federated services.

2. Presentation Layer

As mentioned above, FIPER is based upon a web-centric and web-aware architecture. Data access and sharing are initiated from thin web clients. The model-view-controller (MVC) design pattern is used to design modular and interactive user interfaces. Each of the three components of the MVC design is explained in more detail below.

Models are those components of the system application that hold application data. They are the domain-specific software simulation or implementation of the application's central structure. Models implement application functions that handle data-flow computation in response to user interaction sent from the controller component [7,8].

Views deal with everything graphical. They request data from the model and display data. They may also contain subviews and may be contained within superviews. Views are closely associated with a controller. Each controller-view pair has one model, whereas each model may have many controller-view pairs.

Controllers contain the interface between the associated model and the views and input devices. They also deal with scheduling interactions with other controller-view pairs. The controller tracks mouse movement between application views, implements messages and button activity, and implements input from input sensors.

The relationship between these three components and their interaction is depicted in Figure 3. As illustrated in Figure 3, an MVC triad is intimately connected. In particular, the view knows explicitly about the model and the controller. The controller knows explicitly about the model and the view. However, there is no explicit connection from the model to the other two. This implicit connection between models and view/controllers allows the model to be separated from the application. Therefore, different controller-view pairs can use the same model.

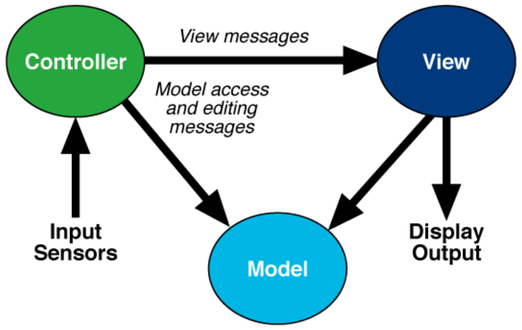

Figure 3: The model-view-controller (MVC) triad, allowing modular component plugin

The architecture diagram for the presentation layer is presented in Figure 4. The MVC approach is used throughout this design.

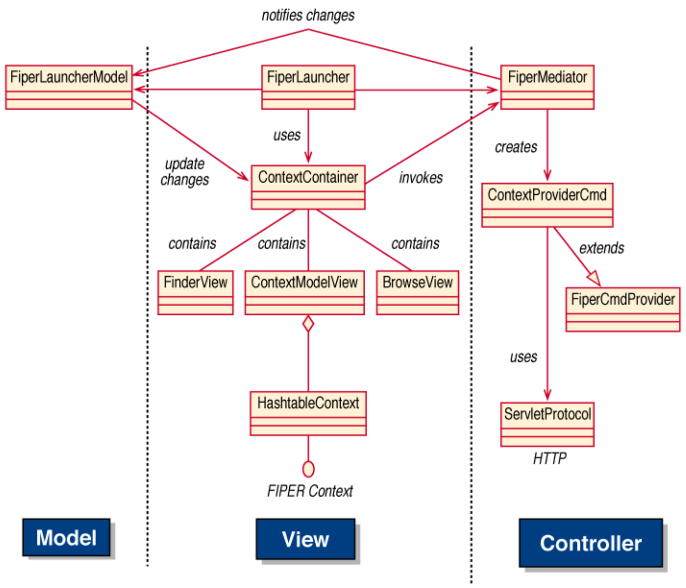

Figure 4: Example showing how the MVC paradigm is applied in the modular FIPER presentation architecture

The class FiperLauncherModel, which is extended from Java object Observable, represents the model here. The Observable class represents an observable object, or data in the MVC paradigm. An observable object can have one or more observers. An observer may be any object that implements the interface Observer. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by calling their update method.

A view class is extended from the Java Panel object, and implements the Observer interface. The name of each view object can be self-explanatory for one's purpose. The class ContextModelView is to display the context model as a tree structure. Control buttons are provided for the user to manipulate the context model view. Detailed information on the usage of each button is described in the next section. To display a tree structure, a TeaSet widget called Forest is used. TeaSet Widgets [11] are a large collection of high-performance, low-resource-consuming Java GUI components that help Java developers to create sophisticated user interfaces within resource-limiting environments. The classes FinderView and BrowseView support search and browse functions. In FIPER, a context is grouped into domains and subdomains. Each domain contains several subdomains. Each subdomain contains contexts. The action flow is as follows:

Step 1: A domain is selected; a change notice is sent to the model by calling notifyObservers

Step 2: The model sends an update back via the update method; a list of subdomains is displayed

Step 3: A subdomain is selected; a change notice is sent to the model by calling notifyObservers

Step 4: The model sends an update back via the update method; a list of the context is displayed

Step 5: A context is selected; a change notice is sent to Model by calling notifyObservers

Step 6: The model sends an update back via the update method; the context model is displayed as a tree structure

Each view object has a model field that points to FiperLauncherModel. Since FiperLauncherModel is an Observable object, each view object has to add it-self as the model's observer by calling the addOb-server method.

After adding itself to observers of the model, each view object needs to implement the update method in the Observer interface in order to process updates. The actual implementation of the update method depends on the functionality of each view object. For instance, class FinderView should implement a list structure to contain and display the list of context model names. This list should be updated in the update method to contain the updated data from the model. Meanwhile, class ContextModelView should have its Forest object updated, to contain the changed data.

Class FiperMediator is the Controller here. It controls the data change request from a view and transfers it to the model. Moreover, it invokes user commands by instantiating the ContextProviderCmd object. Class ContextProviderCmd is responsible for executing user commands and assembly of parameters for database persistence operations. It uses ServletProtocol to establish an HTTP connection with the Persistence layer as described in section III.

Each time the model notifies a change, it processes the data according to the user request. This notification sends an update to all observers. The observers will update for the change, but only for the observers that are interested in the update event. Conditions will be passed on to the observers to determine if the data are needed; if so, the change will be reflected in their views.

3. Business Logic Layer

FIPER supports three centricities and deploys three neutralities [2]. FIPER's three centricities are network centricity, service centricity, and web centricity. A FIPER federation is composed of various service providers; any of these can come and go, and the system can respond to changes in its environment in a reliable way (network centricity). Services in the FIPER environment can discover lookup services and join the federation or lookup for relevant services in order to cooperate in a distributed environment (service centricity). Users can request to use multiple services and check the status of their submissions in different locations through an HTTP portal with thin web clients (web centricity).

The three neutralities FIPER deploys are location neutrality, protocol neutrality, and implementation neutrality (see Figure 5).

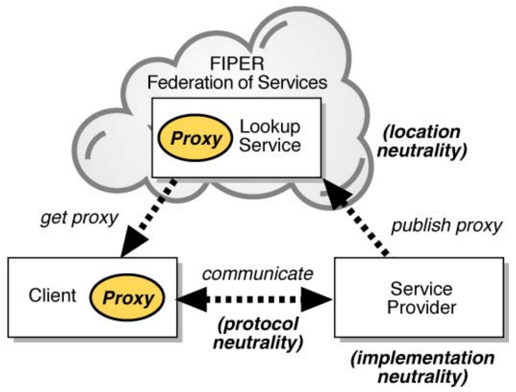

Figure 5: FIPER's three neutralities, providing a simplified, highly flexible software environment

Services need not be co-located; lookup services are discovered and used to find a particular service, which simplifies management of the entire software environment (location neutrality). In addition, the way in which clients communicate with a service provider is not important. Clients are not aware of what protocols are used or where the implementations reside (protocol neutrality). Furthermore, the clients who use the FIPER services do not need to know what languages are used or how a service is implemented (implementation neutrality). In all, FIPER provides accessibility through web-centric architecture, self-manageability using federated services, scalability via network centricity, and adaptability with the power of mobile code inserted for execution through service providers.

To be able to achieve the network-centricity and service-centricity requirements, we choose to use the Jini™ Network technology from Sun Microsystems [4, 5, 9]. Jini is a set of specifications allowing federations of services on a network and providing a framework that allows those services to participate in certain types of operations. As opposed to server-centricity, where all the data and services are located in a specific or predefined server, network-centricity can allow many services at unknown locations to be found and then executed. Service providers are found and resolved through a lookup service provider (Registrar). New service providers are added to the Registrar by discovery and join. To publish a service, its service provider first uses a discovery protocol to locate an appropriate lookup service and then joins, or registers, with the lookup service. Services can communicate with each other in the entire federation creating communities of services.

Each service in the network is an autonomous business logic unit that can serve other units and also be served by them. Data persistence functionality is provided by one of the services that exists in the network. This persistence service handles client requests and communicates with the product data repository. It provides concurrent sharing and data access.

The lookup service is similar in principle to the naming and directory server used in server-centric distributed network environments. For each service it holds service attributes along with its proxy. An application that wants to use a Jini service finds the desired service by matching the service's attributes within the lookup service. A Jini service provider must register itself with a Jini lookup service and maintain an active registration in order for applications to find its service.

To locate a Jini lookup service, which itself is also a Jini service, Jini provides three discovery protocols. When a FIPER service provider starts, it discovers all relevant lookup services on the network through discovery. Then it registers with a discovered lookup service by using Jini's join protocol. In addition, the service provider may listen for new lookup services that start on the network and join them if desired. After the service is registered, its provider has to actively manage its relationship with the lookup service. By managing the relationship, service providers have to renew the lease they received when they registered with the lookup service. Otherwise, the lookup service will assume this service provider has gone away and free the resource allocated. Figure 6 illustrates such a service.

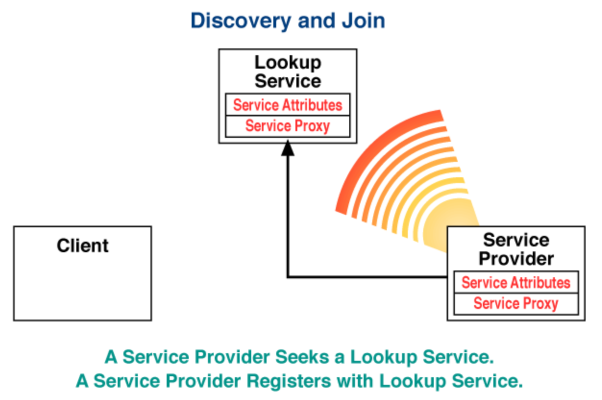

Figure 6: The discovery and join protocols used for registering a Jini service

For a client to be able to use services registered with a Jini lookup service, the client needs to discover the Jini lookup service as well. After the lookup service is located, the client will perform lookup for the perspective service according to its required attributes. The lookup service will pass back a copy of the service proxy to the client, and then the client will communicate with the service provider via the proxy, as illustrated in Figure 7.

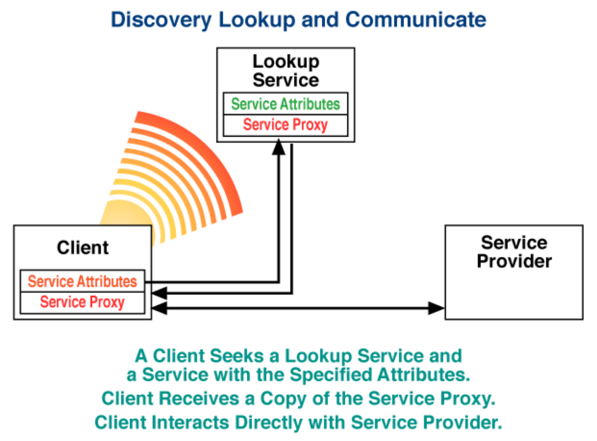
Discovery Lookup and Communicate
Lookup Service
Service Attributes
Service Proxy
Client
Service Attributes
Service Proxy
Service Provider
A Client Seeks a Lookup Service and a Service with the Specified Attributes.
Client Receives a Copy of the Service Proxy.
Client Interacts Directly with Service Provider.

Figure 7: The discovery lookup and communicate protocols used by clients of Jini services

4. Repository Layer

Data persistence is the ability to have information from an application instance exist for later instances of that application or even other applications to use. More often, we represent data that need to be persistent as a business object, which can retain its state and integrity while being concurrently accessed. A business object often uses data that needs to be persistent, and encapsulates the business rules the data obey. One of the major problems for the development and usage of business objects is their need for persistency. Moreover, services that constitute application business logic also need to store their data. The data along with the objects have to be available for concurrent access, while the integrity of their state is maintained. The persistency has to be provided in a way that supports efficient programming and the development of maintainable systems.

Many factors can affect the data persistence ability. The three most important factors are transparency of persistence, data integrity, and service availability [6].

The transparency of persistence means that the environment has to take care of the externalization process. In case of failure, a consistent state has to be reconditioned by this environment. Many occasions require a decision about whether the business object data really has to be made persistent or if temporary changes could be thrown away. Furthermore, users have to be informed about the persistency status. All issues on the transparency of data persistence have a strong impact on the presentation component of a business object capsule, which can follow various models to represent the changes in persistent and temporary object data.

Data integrity is also a big concern in data persistence. In a distributed environment where several users concurrently access multiple persistent objects, a synchronization mechanism must be provided to ensure database integrity, shareability and recovery. When performing a database transaction, ACID (Atomicity, Consistency, Isolation, Durability) properties have to be satisfied.

In the FIPER environment, all services are published as Jini services, including data persistence services. The last factor concerning the availability of a persistence service is that it is critical for other FIPER services sharing data across engineering jobs.

To design the data persistence layer, we apply the Java database persistence design pattern. In this design pattern, all the objects that need to be saved into the database and be persistent are identified as Persistent objects [3]. In the FIPER sense, context models are represented as Persistent objects. In this design pattern, there are five core elements that form the essence of any persistence paradigm, as described below.

A **Persistent** object can construct itself based on the data returned from the database, and be able to keep track of its modification states. The Persistent object contains no code specific to any data storage type, so those plug-in modules can be provided for various data storage technologies.

A **PersistentPeer** is designed as a Java interface, which contains various database operation prototypes. A PersistentPeer is used to determine how an object becomes persistent. The actual implementation of PersistentPeer interface will be according to the underlying database technology. Thus, when a persistent object wants to make itself persistent, it only needs to call the assigned PersistentPeer implementation to perform the storage task. This design approach makes the persistent object flexible for any storage technology given that the PersistentPeer implementation of that technology is provided. For example, if JDBC is used, then a DatabasePeer object is used for relational databases.

In order to share data in a concurrent manner, while preserving the integrity of the data, a **Lock** object is responsible for providing a locking mechanism to prevent data corruption from concurrent modifications.

There is an abstract class called **Transaction**. The class Transaction is implemented by different persistence packages for managing persistence operations on groups of objects. A Transaction gets created based on a data store URL that tells the Transaction object what type of data store it is dealing with, as well as where that data store is. The **DatabaseTransaction** class implements the Transaction interface and serves as a wrapper around the JDBC Connection class. It allows peers access to the JDBC Connection in addition to triggering commits and rollbacks as needed by the persistence package. All Peer implementation objects should use the DatabaseTransaction class to perform database operations.

In case of adopting new database technology, the PersistentPeer implementation class and Database-Transaction class need to be altered accordingly. In Figure 8 the UML class diagram depicts the design for the persistence layer.

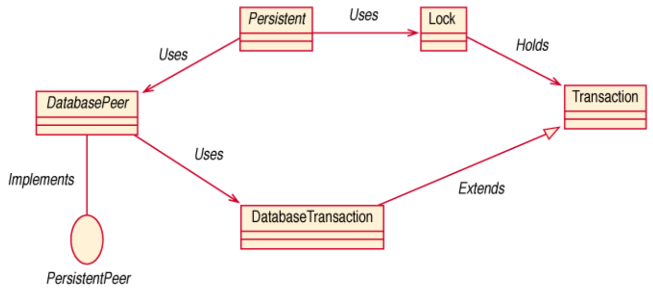

Figure 8: Core persistence architecture (compare to the extended FIPER design, Figure 9)

The persistence design pattern lays out the fundamental structure for building a persistence layer. For the specific needs of the FIPER context model, this basic framework needs to be extended. The complete FIPER persistence layer design is presented in Figure 9.

Following is the description and purpose of each of the new introduced classes in this persistence paradigm.

The class Mandate is responsible for carrying the user command for invoking database transactions from client to the FIPER Cache. FiperLauncher is one of the view objects in the MVC paradigm. The Mandate contains variables to distinguish different database operations. A database operation result is also carried back via the Mandate to the client. Thus, the Mandate serves as a messenger between the web client and CacheServer. For each database transaction, an instance of the Mandate object is created at the client side. This Mandate instance is passed to the CacheServer via ServiceServlet through HTTP protocol. After the transaction is completed, the same Mandate instance will be passed via ServiceServlet with the transaction result. The client then will retrieve the result from the Mandate instance for further operations.

ServiceServlet receives a Mandate object from the client over HTTP then locates the CacheServer, which is the data persistence service provider. ServiceServlet can locate the CacheServer object either through remote method invocation (RMI) or Jini discovery.

The class CacheServer handles all persistence layer operations. Before starting a persistence operation, a Mandate object will be passed to the CacheServer via ServiceServlet. At this stage, the CacheServer will decode the properties of the Mandate object to get the respective user command and arguments. Depending on the user's command, the CacheServer performs preparation procedures to start the persistence operation. After a persistence operation is completed, the CacheServer retrieves the result from respective Persistence objects, assembles the result to the Mandate object, and then sends the Mandate object back to the client via the ServiceServlet. To ensure isolation and atomicity properties for keeping data integrity, a Transaction is always invoked from the CacheServer, so that each Transaction is independent from others, preventing interactions from occurring. The CacheServer is implemented as an RMI object that can be used as an RMI server. Also the CacheServer can be published as a Jini service using the FiperJoiner utility class.

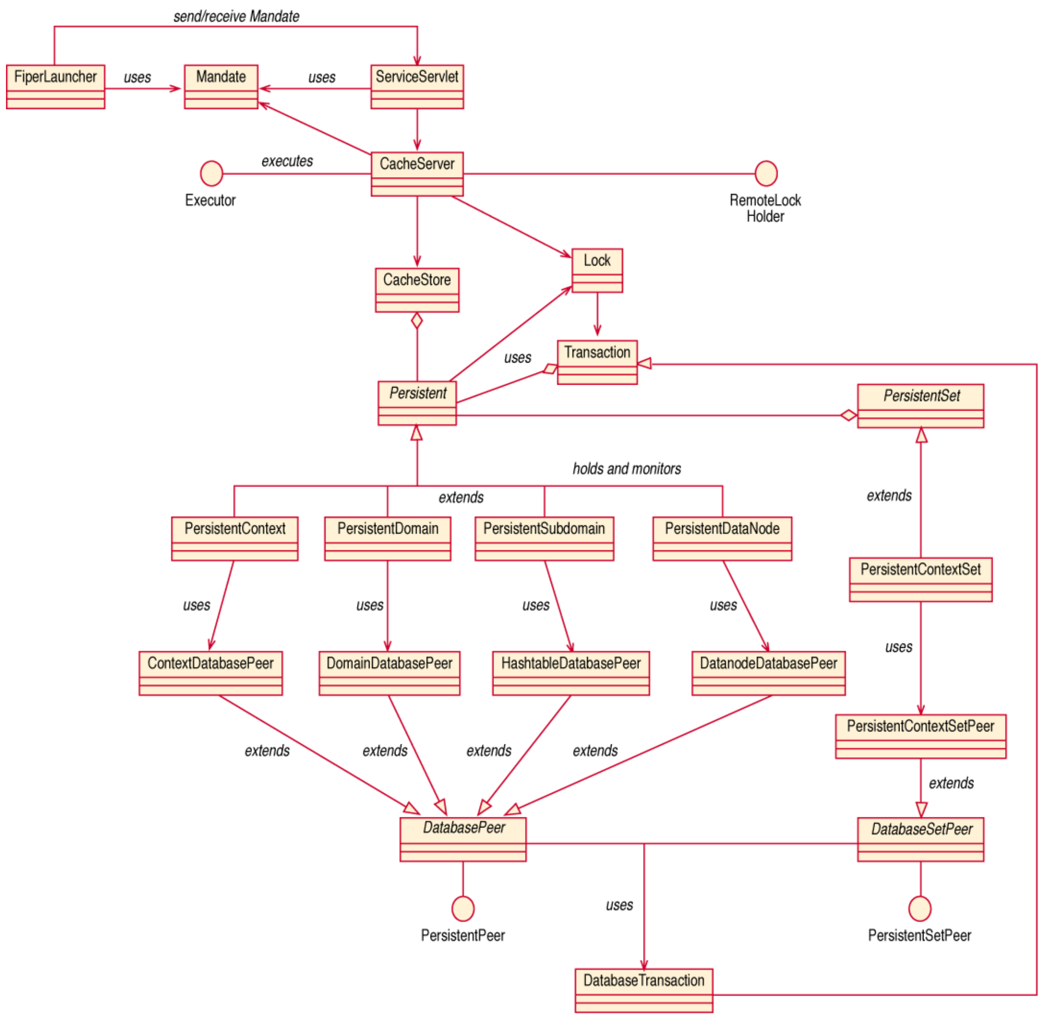

Figure 9: FIPER persistence layer architecture, providing the flexible, robust services required for web-enabled network applications

The CacheStore class is a repository for storing Persistent FIPER objects that are restored by previous transactions. Its purpose is to reduce the traffic of accessing a database by web clients, and to reduce the overhead of retrieving the same data over and over again. It serves as the middle object repository from web client to the backend database repository. After the CacheServer invokes a transaction, it will first send the Transaction object to CacheStore to see if the Persistent object is already there. If so, that Persistent object is retrieved and processed. Otherwise, the backend database transaction will begin. Every new Persistent object that's retrieved from the database is added to the CacheStore. Persistent objects stored in CacheStore are synchronized with the database, so that a change to a Persistent object will result in a change to the corresponding data in the database.

The classes PersistentContext, PersistentDomain, PersistentHashtable, and PersistentDatanode are all subclasses of the Persistent class. As mentioned above, they all inherit the persistent behavior defined in the Persistent class. Additional code is added to implement various features for each individual Persistent object. The corresponding Peer classes provide the implementation details using JDBC to perform the actual database operations. They all inherit from the DatabasePeer class and implement the PersistentPeer interface as defined in the persistence design framework.

5. Conclusion

Users of the FIPER environment can effectively create, edit, and share context models through thin web clients. After the user action is invoked, it will be passed on to the business logic layer, where the concurrent data sharing mechanism resides. Through discovery/lookup, the service provider for data persistence can be located and used for concurrent access. In order to preserve the data integrity and persistency, the backend database is accessed through a data persistence layer, which is responsible for maintaining FIPER business objects independently of the database used. At this stage, a business object is returned to the business logic layer to perform other services or presented to the user for further modifications or to perform engineering tasks.

6. Acknowledgments

This research is jointly funded through the National Institute for Standards and Technology—Advanced Technology Program (NIST-ATP) and the General Electric Company. The authors would like to acknowledge this support, as well as the valuable input from the entire FIPER team.

References

[1] FIPER project information: http://www.oai.org/FIPER.html http://www.crd.ge.com/eml/whatwedo/fiper/index.hml http://www.atp.nist.gov/www/comps/briefs/99013079.html
[2] Röhl, P.J., Kolonay, R.M., et al., A Federated Intelligent Product EnviRonment, AIAA-2000-4902, 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, September 6-8, 2000.
[3] George Reese (1997), JDBC and Java, O'Reilly & Associates, ISBN 1-56592-270-0.
[4] Scott Oaks, Henry Wong (2000) JINI in a Nutshell, O'Reilly & Associates, ISBN 1-56592-759-1.
[5] W. Keith Edwards (1999), Core JINI, Prentice Hall, ISBN 0-13-014469-X.
[6] AdvancedConcent, Inc. (1999), "White Paper Java Persistency Framework," http://www.advanced.ch/Resources/White_Papers/Java_Persistency/body_java_persistency.html
[7] Cristobal Baray (1999), "The Model-View-Controller (MVC) Design Pattern," http://www.cs.indiana.edu/~cbaray/projects/mvc.html
[8] Steve Burbeck (1997), "How to Use Model-View-Controller (MVC)," http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html
[9] Bill Vernners (1999), "Jini: New Technology for a Networked World," http://www.sunworld.com/swol-06-1999/swol-06-jiniology.html
[10] Wilf R. LaLonde, John R. Pugh (1991), Inside Smalltalk, Prentice Hall, ISBN 0-13-465964-3.
[11] "TeaSet Widget," InetSoft Technology, http://inetsoftcorp.com
[12] M. A. Rosenman, J. S. Gero, "Purpose and function in a collaborative CAD environment", Reliability Engineering & System safety, Vol. 64, pp. 167-179, Elsevier Science Ltd
[13] M. A. Rosenman, J. S. Gero, "CAD Modeling in multidisciplinary Design Domains", Key center of Design Computing, University of Sydney (internal report), 1999, pp. 335-347
[14] M. A. Rosenman, J. S. Gero, "Modeling multiple views of design objects in a collaborative CAD environment", Computer-aided design. 1996, Vol. 28, No. 3, pp. 193-205, Elsevier Science Ltd.
[15] R. W. Amor and J. G. Hosking, "Multi-Disciplinary Views for integrated and Concurrent design", Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand.
[16] Teresa De Martino, Bianca Falcidieno and Stefan Habinger, "Design and engineering process integration through a multiple view intermediate modeler in a distributed object-oriented system environment", Computer-Aided Design, vol. 30, No. 6, pp. 437-452, 1998 Elsevier Science Ltd.