11

DATA INTEGRATION

ENTERPRISE APPLICATION/INFORMATION INTEGRATION (EAI/EII) IN GRID

In the chapter on Web Services, we will discuss an evolution to Web Services, starting from the point-to-point integration of standalone systems, to the client/server topology, to distributed computing [straight-through processing (STP)] and finally to the grid topology and the compute utility. This is a long progression of new and very different compute topologies that ushered in their own distinct operational environments, creating an intertwining that exists today. To better understand information integration within the grid, let us take a brief look at the evolution of enterprise information integration, better known as EII.

Straight-through Processing (STP), EAI, and EII

As client/server evolved into distributed computing, new buzzwords emerged. In the early days, enterprise application integration (EAI) was very common, and as time went on, achieving straight-through processing (STP) by leveraging EAI became the trend. Today, the more commonly phrase *enterprise information integration* (EII) has come to the fore. We would like to provide a level of understanding associated with each of these three commonly used acronyms:

- *Enterprise Application Integration* (*EAI*). The enabling of data sources and applications to communicate with each other via a network without custom

Distributed Data Management for Grid Computing, by Michael Di Stefano
Copyright © 2005 John Wiley & Sons, Inc.

process and point-to-point connectivity software, often referred to as a "spaghetti mess." The resulting infrastructure replaced the spaghetti mess with "middleware pipes." Through middleware, information flows among the applications throughout the business units of an enterprise.

- *Straight-through Processing* (*STP*). Like EAI, STP is designed to provide end-to-end business processing automatically and with little or no manual intervention. Each application sends the information to an infrastructure that allows for data processing, including data extraction, data parsing, data manipulation, and data reformating. In addition, this infrastructure is required to provide intelligent data routing and business processing to whatever end system or application requires the data. This infrastructure, or hub, as it is commonly termed, will provide the downstream applications with the data that they require in an automated fashion.
- *Enterprise Information Integration* (*EII*). The purpose of EII is to provide access to data from multiple sources, making the request transparent to the application. Thus, the data are automatically aggregated from the various sources and the requesting application does not have to deal with determining which source will provide which data, and with connecting to each source directly. EII allows all back-end information to be seen as if it came from one comprehensive, global database.

STP and EAI tend to go hand in hand; application integration is required to achieve a near-real-time enterprise or STP. The progression to STP is a direct response to various business drivers:

- In the financial industry, STP was driven by the initiative of moving trading to clear in one day, a process commonly referred to as $T + 1$.
- Too many manual processes, which increased costs, errors, and processing time.
- The high cost of implementing computer systems.
- The high cost of maintaining computer systems.
- The business shifting focus to a customer service model.

Prior to the emergence of EAI and STP, the predominant system architectures were "stovepiped" (see Figure 11.1), designed to run independently with no interaction. Examples are inventory control, human resources, and sales automation. Over time, more and more lines of business required these systems to share information, to get systems to share tasks and data and to eliminate the need for custom code that is normally written. Typically, these were one-off efforts quite often duplicated by different development groups supporting their respective business units. This normally resulted in many versions of code performing the same or very similar tasks and functions. Maintaining such point-to-point communication was very costly and yielded very few functional benefits. Companies started looking at product solutions that would solve these problems and eliminate the custom development that was in place.

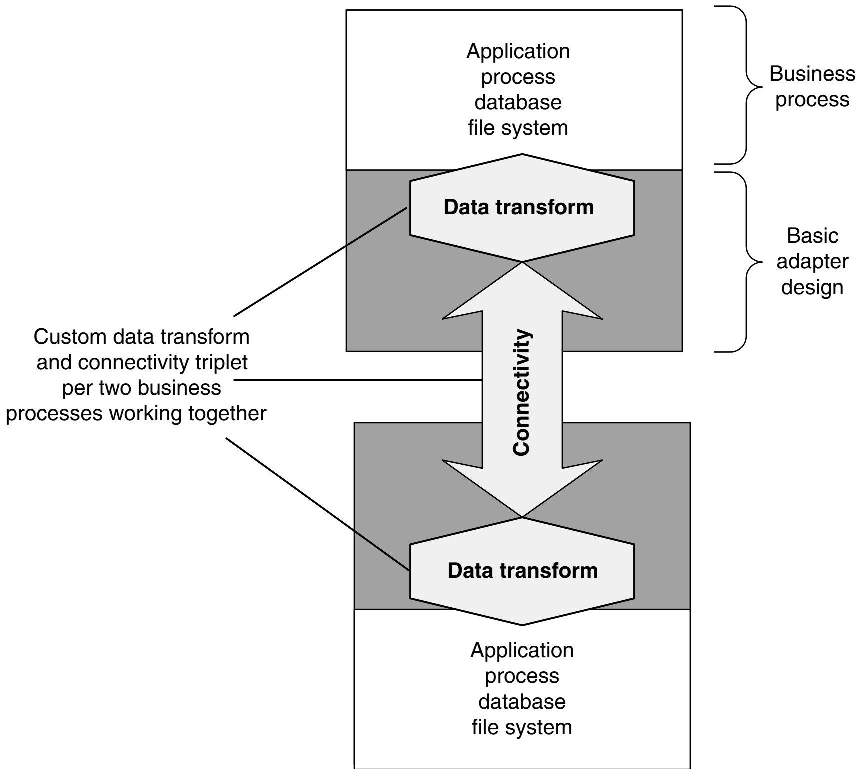

Figure 11.1. "Stovepiped" interprocess and intersystem connection architecture.

What began to emerge was a common process of separating how the applications communicated, for involving data transformation and data representation. Connectivity encompassed the physical hurdles of how the two processes or systems connected to each other, which was typically via a network socket or through file sharing, a more common process than one would expect. The second layer to connectivity was the protocol. The protocol defines the logical interaction between the systems, things such as how to establish a connection, how to terminate a connection, message headers and footers, message counting, how to identify missed messages, and how to request a retransmission of missed messages. In regard to file sharing, the functions included the FTP function, file checksums, and file ready for processing flag. This process employed specialists for each system, network programming specialists, and reams of specifications documenting every aspect of the process.

In conjunction with the system protocol were each system's data formats. This included message bodies, message headers and footers (separate from the communication protocol headers and footers), and field definitions. The message formats ranged from some delimited format (the delimiters could be anything that the developer of the system desired, typically a comma or any other character that was not part of the information being sent), or an offset of field bit position and

size (analogous to a pilot's method of dead-reckoning navigation of an aircraft). The developer knew the starting point of each field since specific sizes were sizes associated with the fields. By counting fields and field sizes, the developer had a good idea of where the next field started and ended.

This process needed to be repeated for each system pair that wanted to communicate and share information and process. The end result was a rat's nest of intertwining systems that cost more to operate and maintain than did the respective systems themselves. As you can imagine, system maintenance and regression testing were a nightmare; the smallest change in one system impacted other systems and caused a cascading regression test and QA (quality assurance) cycle for all other connected systems.

The business driver of shifting to a customer-focused view requires the ability to deliver business processes to the consumer quickly and efficiently. Stovepiped data centers and applications integrated via point-to-point custom code do not lend themselves to this level of business delivery, thus leading to new techniques of system and application integration. This began the evolution to enterprise application integration (EAI).

The architecture for EAI and STP (see Figure 11.2) deals with the abstraction of system conductivity. Systems that interact via the sharing of events and information should be able to publish events leading to the sharing of information associated with the respective events without having to worry about direct point-to-point communication of any type, such as sockets. This concept describes one of the core functions of what was termed "middleware."

The EAI and STP architectures simplified matters, eliminating the complexity of connectivity by ushering in a new technology (and family of jargon) of middleware. The generic definition of middleware is software that connects two separate applications. It is sometimes referred to as "plumbing" or the "glue" that holds or connects applications together and passes data between them. In practice, middleware performs some wondrous feats. It provides a standard method and protocol for all applications to communicate; it completely disconnects any one system from all others that need its information to perform its own tasks; conversely, it can get to any other systems' data that it may need. This disconnect of systems allows for system maintenance without adversely affecting every other system in the information sharing chain simply by requiring a QA test with the middleware product. Note that the data side of the system integration equation is still not addressed. This is where we start to get into the wide variety of and differences in middleware products. At the highest level, there are basically two types, messaging-oriented middleware (MOM), and Common Object Request Broker Architecture (CORBA). The former has an entire family tree of brothers, sisters, and distant cousins, while the latter is attempting to fill the broader scope of service-oriented architecture (SOA), which has seen better days.

Messaging middleware has three basic flavors: simple queuing, where applications have "well known" inbound and outbound queues that anyone can access; message routing features that will automatically deliver "published" messages to all "subscribers" to that message; and data translation tools that will translate data

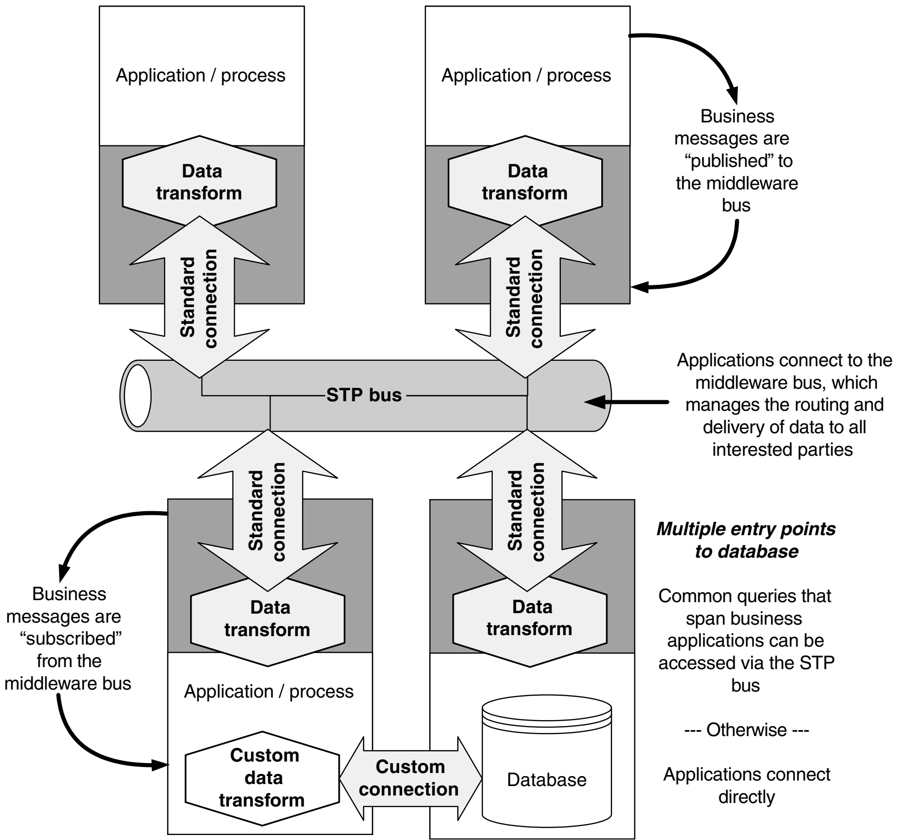

Figure 11.2. EAI and STP architectures.

from the originator's format to the recipient's format while it transports the data between the two systems.

CORBA, on the other hand, encapsulates systems as a "service" and publishes its services for other systems to access. The services include both data and function. Data representation in a CORBA environment is common to all applications to understand independent of hardware platform, implementation, and operating environment. CORBA's data representation is accomplished by using the Interface Definition Language (IDL). A service "publishes" its interface by defining it in IDL and then compiling it for source-level inclusion into the implementation of the service code. Any CORBA service can be "located" simply by making a request to the CORBA service broker. The requesting application has to know something about the service it needs and through a series of inquiries can find out all the details of the service in order to use it. As you can see, it can get quite complicated, and we are only skimming the surface.

CORBA's complexity and closed nature ultimately have led to its filling niche markets only. However, the base technology and lessons learned have led to

today's current generation of service-oriented architecture, namely, Web Services (see Figure 11.3 for a comparison of the two architectures).

As you can see, all the different versions and flavors of MOM and CORBA still result in a complex network of stovepipes of middleware integration (see Figure 11.4) since each vendor's software did not communicate to the other vendor software. Without industry standards, connecting these stovepipes together still required tremendous effort. That and consulting costs were the main reasons for middleware vendor failure. Consulting costs were very high, and custom development was not eliminated even though the applications did not have to worry about such tasks.

One of benefits of EAI is the ability to achieve zero latency, the real-time enterprise via a methodology called *straight-through processing* (STP).

Figure 11.5 shows the evolution from point-to-point to EAI/STP, the service-oriented grid infrastructure.

It quickly became apparent that in order to achieve STP, a second front had to be opened up on information integration: the evolution of enterprise information integration (EII). Without the ability to represent the data independent of their source, the full benefit of STP and EAI architectures cannot be realized. This leads to fulfillment of the second of the two main components to system integration data. The fundamental concept of abstraction of connectivity and data is evident in the service-oriented architecture. SOA is flexible to adapt to and manage process-level and data-level integration, or, as we have been discussing in this book, the compute grid plane and the data grid plane. Grid technology is the evolution of middleware; it is the evolution to the distributed computing environments given birth to by EAI, STP, and EII.

EII IN GRID

Data integration with grid computing builds on the concepts of EAI and EII, which we have touched on in an earlier section. The data grid plane provides the

Web Services		***CORBA***
Open standard	**Service location**	CORBA-only API
Loosely bound	**Service binding**	Tight bindings
Internet standards	**Connectivity**	TCP/IP sockets
XML (open standard)	**Data representation**	IDL (complex process)
Internet standards	**Vendor support**	CORBA vendors interoperate
Simple to moderate	**Ease of use**	Complex

Figure 11.3. Web Services versus CORBA.

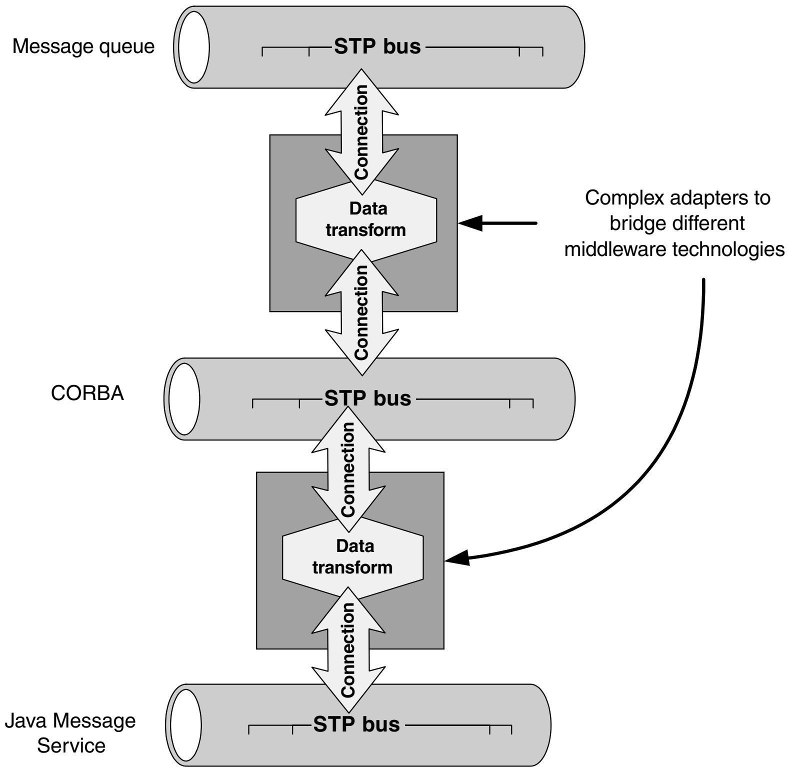

Figure 11.4. Integration of middleware.

focal point for data integration at the business service level. The core principles for data integration in the data grid leverage the same "adapter" techniques from STP; systems join the data grid in methods similar to those which they would attach to an STP bus.

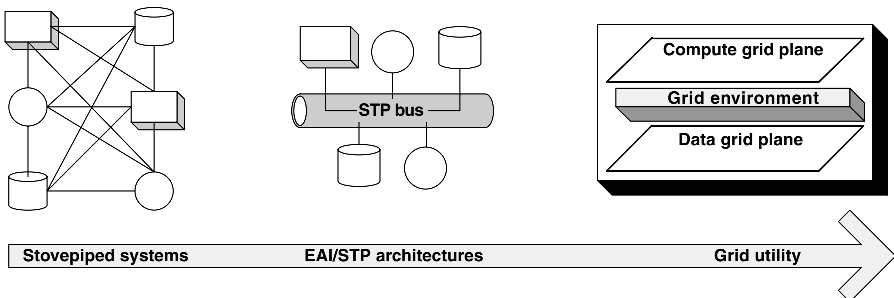

Figure 11.5. Integration architecture evolution.

One of the primary hurdles that had to be overcome in the EAI and EII evolution was the integration of legacy systems, numerous systems that were built across a long timeline. The people responsible for creating these systems—managers, architects, and programmers—have most likely migrated on to different groups or organizations. The technology on which these legacy systems were built can be different from the ones we are using today, technology that may not even be supported by the vendor in the form in which it is used. How many systems have we come across that are compiled against a third-party package, operating system, or an in-house library that is no longer supported because upgrading one vendor's version release will cause integration failure or conflicts with other packages also linked into the system? The inherent knowledge that is still part of the original project team may have migrated with them in their advancing careers. Any system documentations most likely do not capture all the nuances of the system, or the documentation itself may be lost. The investment in time, resources, and cost in the adoption of EAI, STP, and EII cannot be pushed aside, but rather leveraged in order to foster quick adoption of the grid technology and movement to service-oriented offerings. The adapter methodology, technology that is tested and working in production at data centers, must be reused. The methodology for bringing the systems into the STP-EII environment applies to grid computing and data grid integration.

We will see that the evolution of EII into the data grid goes beyond the mechanics of EAI, STP, and EII simply for data integration. In order to provide services in a quick and flexible manner, data management policies must be in place to describe and manage data load and data store: data load policies for the import of data into the data grid plane as well as the data store policies for the data export out of the data grid plane. These data load/store policies must orchestrate with the other policies such as data synchronization policy. We will build on this relationship, which has been introduced in earlier chapters.

EII within the data grid plane adds a layer of abstraction so that data movement decisions are policy-driven rather than being programmed into the business applications and adapters that attached them to the data grid. We will look at the architectures of data load and data store as well as the interaction between these policies and the other data management policies of distributed data management.

Natural Separation of Process and Data

Grid computing offers a natural separation of the process and the data. The compute grid plane manages the execution of the business logic of a process, service, or application. The data grid plane manages the data access, distribution, quality of service, and availability of the business data used by the business logic executing in the compute grid plane.

Let us start from the perspective of a developer—specifically, a developer using object-oriented methodologies to implement a system. This is simply a tool enabling us to visualize the separation of process and data management. Applications written in non-object-oriented paradigm follow the same separation of process and data separation in the grid. The point is more easily visualized in the following example.

This is a fair assumption as most applications written since the mid-1990s use this paradigm. Java, C++, C#, and SmallTalk are object-oriented programming languages, so the assumption is that if your application is written in one of these languages, it uses object-oriented design principles. This statement may make some people's hair stand on end, as C++ does not enforce object-oriented principles. This is a topic of discussion for another time. Please allow this indulgence for the scope of this discussion.

The structure of business objects takes the form of methods and attributes. *Methods* are the program or the business logic implementation of the object. The *attributes* are the data with which the business logic operates in order to perform its function. The following pseudocode shows a typical declaration of a business object:

```
1  public class FooBar
2  {
3  //NOTE TO THE READER
4  //A little Object Ease, Anything that is Public anyone can
     access. Anything that
5  //is Private ONLY this business object can access.
6   //
7
8   //===== Declare the Business Data Attributes of the
    Business Object =====
9   //declare the object's Private Attributes
10  //
11  private String thisProcessName;
12  private Integer thisProcessState;
13
14
15  //===== Declare the Business Processes (Methods) of the
      Business Object =====
16  //declare the object's Public Methods
17  //
18
19  //===== Public Method - Constructs the Object =====
20  public FooBar()//Object Constrictor - used to put the
    object in a well
21                 //known state at its inception
22  {
23    thisProcessName="FooBar";
24    thisProcessState=0;
25  }
26
27  //===== Public Method - Change State =====
28  public void ChangeState()//Simple add one to the counter
29  {
30    thisProcessState=thisProcessState+1;
31  }
32
```

```
33  //===== Public Method - Show the Business Process's State
      =====
34  public void HelloWorld()//prints out the object's name and
    state
35  {
36  print(''Process Name is:''+thisProcessName);
37  print(''Process State is:''+thisProcessState);
38  }
39
40  }//End Object Decloration
```

As is seen in this example, the data attributes are declared separately from the business logic methods; however, the data are used, read, and modified in the business methods. In a traditional compute environment, where *FooBar* executes on a single machine, we do not think of this as two separate compute and data management environments. The operating system creates a separate processing and memory space for the executing program and manages both for us. In a grid environment, it is the compute grid plane that acts as the grid operating system for resource and process execution and the data grid plane serves as the data management system for the grid operating environment.

The compute grid plane keeps track of the compute nodes capable of executing our business process *FooBar.* It also knows which of *FooBar*'s methods need to be executed. It will make the best possible match of task and resource; therefore, it can execute the "HelloWorld" method on compute node 1001 and the "ChangeState" method on node 1275.

It is the data grid plane's job to ensure that the data attributes of *FooBar* are available and in a consistent and correct state, accessible on both nodes 1001 and 1275 when needed.

Figure 11.6 shows the natural separation of compute and data management within a grid.

Note that we will reference the *FooBar* code example throughout the remainder of this chapter.

Data Load Policy

In Chapter 9, we discussed the basic principles of both data load and data store policies. Here, we will tackle the mechanical and operational aspects in data integration and EII of the data grid through the data load and store policies.

From the application's perspective, there are two ways to load data into the data grid. First is the do-it-yourself approach. In the *FooBar* example, *FooBar's* constructor initializes the state of the business object by setting the name to *FooBar* and stage to zero. In the *ChangeState*( ) operation, it modifies the state by adding one to the data attribute. Basically *FooBar* loads and changes its data attributes itself. Specifically, it loads in static values into the attributes. However, it could have just as easily have connected to a database, queuing system, or file system to load in the value on construction or leveraged some other external data source to change its state.

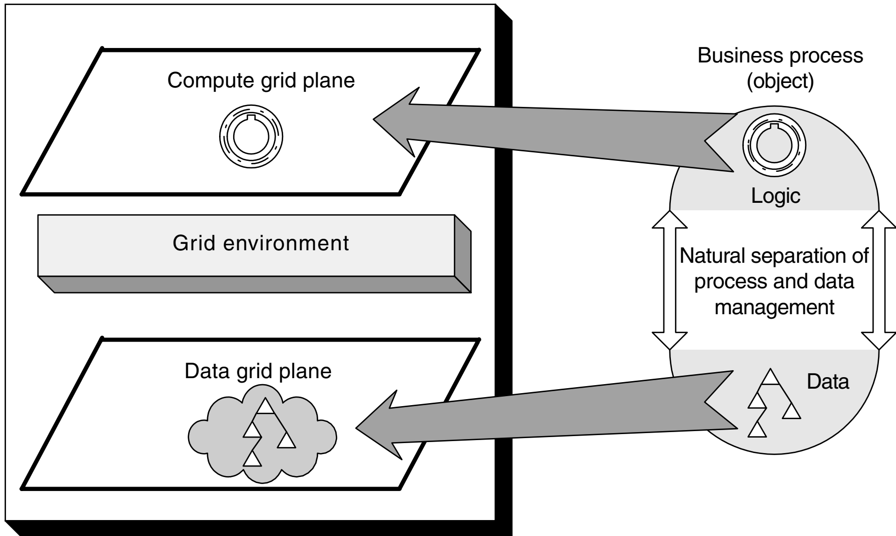

Figure 11.6. Natural separation of process and data.

The data load policy comes into play when *FooBar* relies on the data grid for loading some or all of its data attributes for it. For example, we can establish a data load policy for *FooBar* as follows

$$DataLoadPolicy = DLP\left(\begin{array}{l} FooBar_DLP, \\ ExampleRegion, \\ Granularity(Grouping(1), Frequency(500)), \\ FooBarFileAdapter(\,) \end{array}\right)$$

with a data synchronization policy of

$$SynchronizationPolicy = SP\left(\begin{array}{l} FootBar_SP, \\ ExampleRegion, \\ Scope(Boundary(\text{``}inter\text{''}), List(\text{``}foobar_ProcessStage\text{''})), \\ Transactionality(\text{``}transactional\text{''}), \\ LoadStore(List(\text{``}Foobar_DLP\text{''}), List(\text{``}FooBar_DSP\text{''})), \\ Events(NULL) \end{array}\right)$$

Given the policies set above, *FooBar* can be modified as follows. Obviously this omits details such as establishing a connection to the data grid and using cumbersome get and set methods. The code snippet is for showing the concept only. Working code examples are provided in a later chapter.

```
1  //===== Public Method - Constructs the Object =====
2  public FooBar()//Object Constrictor - used to put the object
   in a well
```

```
3                //known state at its inception
4  {
5   thisProcessName = "FooBar";
6   thisProcessState = dataGrid.get("ExampleRegion",
                         "foobar_ProcessState");
7  }
```

Here, we are creating a "local copy" of the "FooBar" process state contained in the data grid. This local copy can be used elsewhere in the "FooBar" business process and saved to the data grid when necessary.

Let us step through what happens when line 6 of the sample code above is executed. First, we are assuming that we have a connection to the data grid, done earlier in the program, which is represented by the "dataGrid" object. The "dataGrid.get( )" call has two parameters; one is the data region, which in this example is "ExampleRegion." The data region is where the data atom "foobar_ProcessState" and the name of the data atom is re sident, the second parameter to the "dataGrid.get( )" call. Since we have already defined policies for synchronization and data load, the data grid will do the following:

```
if (``foobar_ProcessState'' Data Atom is resident in the Data
Grid)
then check to see if another process has a lock on it
      if (``foobar_ProcessState'' Data Atom is locked)
      then wait till lock is released
      endif
      Read and return value of ``foobar_ProcessState'' to the
        business process
else
    //The Data Atom ``foobar_ProcessState'' is NOT resident in the
    //Data Grid
    //As defined by the Data Load Policy use the Adapter
    //``FooBarFileAdapter()''
    //that knows where the file resides, external to the Data
    //Region, access the file
    //and get the value for ``foobar_ProcessState'', translate it
    //to the proper data
    //representation and populate the Data Atom
    //``foobar_ProcessState''
    Create and Lock the Data Atom ``foobar_ProcessState''
    rawDataFormat = FooBarFileAdapter().load
    //(``foobar_ProcessState'')
    foobar_ProcessState = FooBarFileAdapter().translate
    //(``foobar_ProcessState'')
    Return the value of ``foobar_ProcessState'' to the business
    //process
    Release the lock on ``foobar_ProcessState''
end if
```

The end result of the business process *FooBar* is that the data it needs to perform the function "foobar_ProcessState" are physically retrieved from another data

source, stored in a foreign format in a transactional manner without any working knowledge or code to do so. The entire process is defined by the data management policies and managed by the data grid on behalf of *FooBar*.

Simple changes in the synchronization policy can have a major impact on the behavior of the data grid and *FooBar*. For example, the data load policy can use a different adapter, which will get the data from a completely different source and data representation. Or the data synchronization policy can be switched to nontransactional, which will eliminate the need to lock data atoms, thus speeding up the entire process.

Figure 11.7 shows the separation of process and data management, the interaction of policy and adapters to achieve true enterprise information integration in the grid through the data grid's distributed data management policies and function.

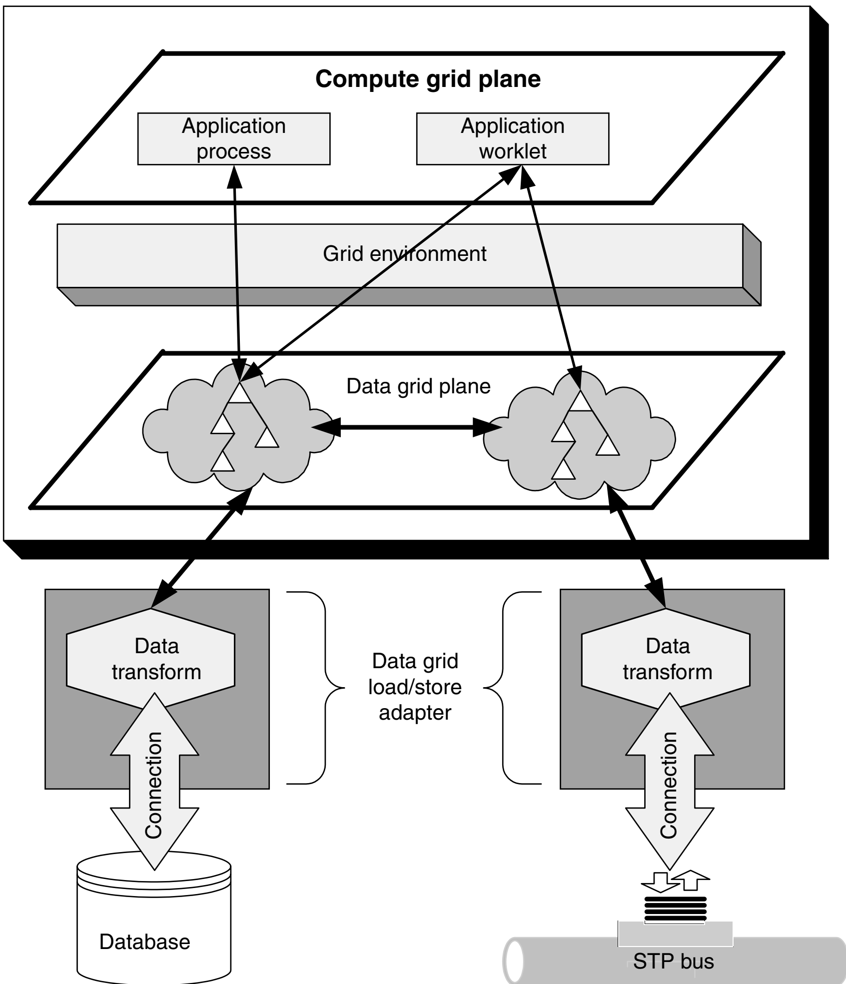

Figure 11.7. EII through data load, data store, and synchronization.

Figure 11.7 shows an application process "joining a data region" that has synchronization policies with a database and a second data region that in turn has a synchronization policy with an STP message queuing bus. Also, there is an application worklet that has joined both data regions. Both the application process and the worklet can share and interact with data sets in the first region without requiring any knowledge of where the data originate from, how the data are represented, or any other mechanical aspect of data region integration.

Data Store Policy

The discussions on data store policy are very similar to those of the data load policy. The differences are evident in the interaction with data sources and the effects of data synchronization policy on the system behavior.

Note, as with the data load process—which has two choices for loading data into the data grid—that an application can only save or store data out of the data grid through its data management policies and procedures. In the *ChangeState*( ) operation, it modifies the state by adding to the data attribute. The result is then saved to the data grid through the *dataGrid.set*( ) operation.

FooBar relies on the data grid to save some or all of its data attributes. For example, we can establish a data store policy for *FooBar* as follows

$$DataStorePolicy = DSP\begin{pmatrix} FooBar_DSP, \\ ExampleRegion, \\ Granularity(Grouping(1), Frequency(500)), \\ Operation(``store"), \\ FooBarFileAdapter(\,) \end{pmatrix}$$

with a data synchronization policy of

$$SynchronizationPolicy = SP\begin{pmatrix} FooBar_SP, \\ ExampleRegion, \\ Scope(Boundary(``inter"), List(``foobar_ProcessStage")), \\ Transactionality(``transactional"), \\ LoadStore(List(``FooBar_DLP"), List(``FooBar_DSP")), \\ Events(NULL) \end{pmatrix}$$

The data store policy as defined above defines the data region as *ExampleRegion* with a granularity of one grouping and 500 frequency. The operation to be performed is a "store" function and the adapter being utilized is the *FooBarFileAdapter*( ).

The synchronization policy again is for the same data region *ExampleRegion* and the *Scope*( ).

Given the policies defined above, the code for *FooBar* can be modified as follows. Note that the code snippet is for showing the concept only. Working code examples are provided in a later chapter.

```
1  //===== Public Method - Change State =====
2  public void ChangeState()//Simple add one to the counter
3  {
4   dataGrid.set(``ExampleRegion'',   ``foobar_ProcessState'',
   (thisProcessState+1));
5  }
```

The "local copy" of the *FooBar* process state is being changed by incrementing the state by one or adding one to it and then being stored into the data grid's data region, *ExampleRegion*'s data atom "foobar_ProcessState."

Let us step through what happens when line 38 is executed. Please note the following:

- First, we are assuming that we have a connection to the data grid, established earlier in the program, which is represented by the "dataGrid" object.
- The "dataGrid.set( )" call has three parameters.
- The data region *ExampleRegion* is where the data atom is found.
- "foobar_ProcessState" is the name of the data atom to set and the value to which it is set.

Since we have defined policies for synchronization and data load, the data grid will do the following:

```
if (``foobar_ProcessState'' Data Atom is NOT resident in the
   Data Grid)
then create the Data Atom ``foobar_ProcessState''

if (``foobar_ProcessState'' Data Atom is locked)
then wait till lock is released
endif
//Place a lock on the Data Atom ``foobar_ProcessState'' so no one
//else can access
//it while the update is being performed.
//As defined by the Data Store Policy use the Adapter
//``FooBarFileAdapter()''
//that knows where the file resides, external to the Data
//Region, access the file
//and save the value for ``foobar_ProcessState'', translate it to
//the proper data
//representation
  Lock the Data Atom ``foobar_ProcessState''
  source_FormattedValue = FooBarFileAdapter().translate
  (``foobar_ProcessState'')
  FooBarFileAdapter().save(source_FormattedValue)
  Release the lock on ``foobar_ProcessState.''
```

The end result to the business process *FooBar* is that the data it needs to perform the function ''foobar_ProcessState'' are physically stored to the data grid and the

transition required to update an external system is performed without any working knowledge or code in the business process. The entire process is defined by the data management policies and managed by the data grid on behalf of *FooBar*. This allows for configurable changes to the policies without affecting any code in the business processes.

Simple changes in the synchronization policy can have a major impact on the behavior of the data grid and *FooBar*. For example, the data load policy can use a different adapter that will save the data to a completely different source and data representation. Or the data synchronization policy can be switched to nontransactional, which will eliminate the need to lock data atoms, thus speeding up the entire process. These kinds of policy changes are external to the business process, thus allowing system changes to occur through the change of policy definitions.

Load, Store, and Synchronization

Interaction of the data load/store policies with data synchronization policies defines the behavior of the data region and the exact QoS level required by the business service that it supports. These policies in combination determine the transactional behavior or transactional depth levels of a data region, which in turn determines the performance of the region.

What are the "depth levels" of a transaction? The data grid, its data regions, and its interactions with other data regions and external data sources have inherent levels of depth. Are the data synchronized in a data region limited in scope to the boundaries of that region (i.e., intraregion synchronization)? This is the first level of depth, internal to the data region. The next level, for example, applies if the data region interacts with external data sources and is transactional; is it transactional to the delivery of the data from the data region to the external source? The next-depth level applies if the transaction has completed accessing the external data source and reported the status back to the data region. Is the data atom/data region interacting with more than one data source for a transaction, and are these transactions processed independently of each other or grouped as a single unit?

Figure 11.8 highlights different depth levels, as an example.

This cascading effect—of pinpointing where the data delivery process occurs within the chain of data sources, and whether the business service is satisfied that the data are delivered—is referred to as the "depth levels" of a transaction. Obviously, the data management policies of a data region can extend only so far down into the behavior of an external source's management of a transaction. However, they can define the upper or closest levels to the business application as it interacts with the data grid.

The data load, data store, and synchronization policies in combination all define whether the data are to be managed in a best-faith delivery, data region transaction delivery, or fully fault-tolerant delivery. Let us look at what each of these data delivery modes are, what types of applications they support, and exactly which parameters in the respective policies affect these desired behavior.

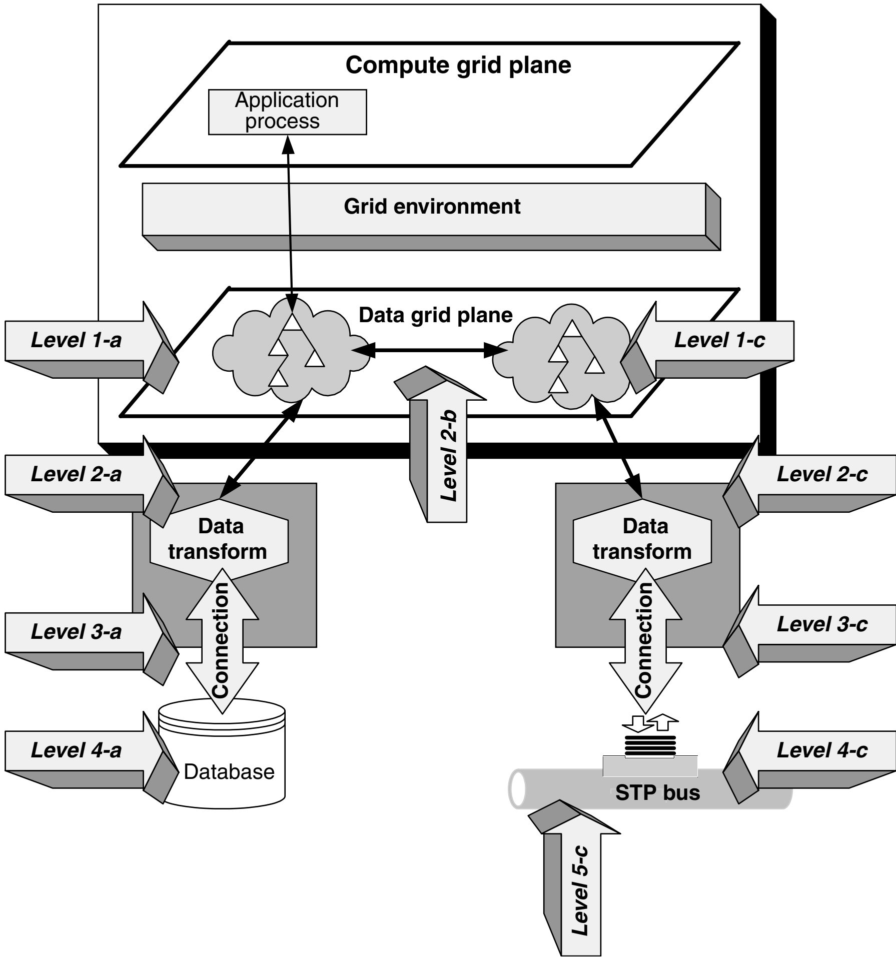

Figure 11.8. Level of depth of a transaction.

Data management policies for load, store, and synchronization from the earlier chapters where expressed as

$$DataLoadPolicy = DLP\begin{pmatrix} PolicyName, \\ Region, \\ Granularity(\,), \\ Adapter(\,) \end{pmatrix}$$

$$DataStorePolicy = DSP\begin{pmatrix} PolicyName, \\ Region, \\ Granularity(\,), \\ Operation(\,), \\ Adapter(\,) \end{pmatrix}$$

$$SynchronizationPolicy = SP\begin{pmatrix} PolicyName, \\ Region, \\ Scope(\,), \\ Transactionality(\,), \\ LoadStore(\,), \\ Events(\,) \end{pmatrix}$$

With the policies defined above, we will discuss the different delivery modes and what they mean:

1. *Best-Faith Delivery*. This is the most "optimistic" of all delivery modes. Here, the business service "trusts" the data grid to deliver the data in its own good time. It is optimistic with respect to the data grid's ability to deliver the data. This mode of delivery is best used by applications that are dealing with data that are time-critical and transient in nature. Examples of the types of applications that require such a delivery mode include Monte Carlo simulations, and the delivery of both news and quote data to trading applications. The data management policies for the synchronization policy for this delivery mode of operation can be set as follows:

$$SynchronizationPolicy = SP\begin{pmatrix} BestFaith_SP, \\ ExampleRegion, \\ Scope(Boundary(``inter"), List(``foobar_ProcessStage")), \\ Transactionality(``nontransactional"), \\ LoadStore(NULL, NULL), \\ Events(NULL) \end{pmatrix}$$

2. *Data Region Transactional*. This is where the data atoms within a data region are transactional with their replicas distributed within the data region. This is important for instances when a level of resilience to hard failures (e.g., failure of compute nodes, or partial network outages) is required but 100% fault tolerance is not essential. In this instance, parts of the data grid can fail but there will be no data loss as part of this mode of delivery. This is best for applications where the volume of data in a region is large and the cost of reload in the case of a failure is too great, especially from an operational window perspective. In addition, applications that are mostly read-only or query-intensive are ideal for this type of mode delivery. Examples are datamart, data warehouse, and OLAP applications. The data management policies for both the load and the synchronization policies in this mode of operation can be set as follows:

$$DataLoadPolicy = DLP\begin{pmatrix} DRTransLoad \\ ExampleRegion, \\ Granularity(Grouping(1), Frequency(500)), \\ DRTransAdapter(\,) \end{pmatrix}$$

$$SynchronizationPolicy = SP\begin{pmatrix} DRTrans_SP, \\ ExampleRegion, \\ Scope(Boundary(``inra"),NULL), \\ Transactionality(``transactional"), \\ LoadStore(List(``DRTransLoad"), NULL), \\ Events(NULL) \end{pmatrix}$$

3. *Fault-Tolerant Transactional.* This is the most pessimistic mode of operation of all the delivery modes. In this mode, the business service has no faith in the data grid's ability to deliver data on its own and must confirm receipt of data delivery for all transactions all the way from the final destination. The data atoms within a data region are completely transactional down to and through the external data source, where the external source confirms that the transaction is complete. The type of applications for which this mode of delivery is best suited is where data delivery is paramount and system performance is not. Examples of applications requiring such a delivery mode are ATMs (automatic teller machines) and accounting and banking systems. The data management policies in this mode of operation for the load, store, and synchronization policies are highlighted below:

$$DataLoadPolicy = DLP\begin{pmatrix} FTTransLoad \\ ExampleRegion, \\ Granularity(Grouping(1), Frequency(500)), \\ FTTransAdapter(\,) \end{pmatrix}$$

$$DataStorePolicy = DSP\begin{pmatrix} FTTransStore \\ ExampleRegion, \\ Granularity(Grouping(1), Frequency(500)), \\ Operation(``store"), \\ FTTransAdapter(\,) \end{pmatrix}$$

$$SynchronizationPolicy = SP\begin{pmatrix} FTTrans_SP, \\ ExampleRegion, \\ Scope(Boundary(``inter"), NULL), \\ Transactionality(``transactional"), \\ LoadStore(List(``FTTransLoad"), List(``FTTransStore")), \\ Events(NULL) \end{pmatrix}$$

Enterprise Data Grid Integration

The buzzwords of *enterprise application integration* (EAI) and *enterprise information integration* (EII) describe how applications and information from various sources can be integrated into a larger, more purposeful, broad, and deep view of an organization's information at a business level. In these types of integration, we are discussing how to manage data within a distributed computing environment, which is just one system within an enterprise. As we have seen in the past with

various database and middleware products, each of these products has its own advantages and disadvantages. On the basis of these advantages and disadvantages, an enterprise will have products from more than one vendor, thus creating the need for EAI and EII. There may be queuing system products from IBM or Tibco, for example. In addition, there may be database products from Sybase or Oracle; therefore, it is reasonable to expect that there will be various data grid products throughout an enterprise. The combinations and permutations of data grids can be as great as we have seen with database/middleware products. The same thing holds true for the compute grid. Some grid vendors may have a metadictionary, while others may be distributed-memory-based. There can be products from more than one grid vendor at any one enterprise implementation. For example, a distributed-memory-based data grid product from company A is used in one area of the organization while a different area can use a data grid from company B. The reason for choosing a respective product within the various areas within the organization is dependent on the business and how the products support the business.

It is reasonable to expect that multiple data grid products will be employed throughout an organization. Therefore, if we do not clearly define an interface for how data grids, specifically data regions within data grids, can interact with each other, then all we will have done is create larger data silos and a chasm that must be crossed in order for those organizations to share data with each other. So let's create a new buzzword; if enterprise information integration deals with the integration of information across an entire enterprise, and if data grid is a single product or single methodology for data integration within a grid, then *enterprise data grid integration* (EDGI) refers to the interoperation of data grids across an enterprise. EDGI is a subset or subcategory of EII. Armed with our new buzzword, let's review some of the data management policies for the data grid:

- *Data distribution policy* describes the distribution of data within the data grid or the distribution of data atoms within a data grid.
- *Data replication policy* describes exactly how the data atoms are to be replicated within a data region.
- *Data load* and *data store* define the mechanics, the adapters for moving data in and out within a data region and the granularity of the data movement process.
- *Event notification policies* notify the registered synchronization policies (and any other registered process for that event) that something—for example, a data atom—has changed state and an action needs to be taken.

The synchronization policy, on the other hand, depends on these policies to perform the mechanics of data movement within and between the data region and external sources. Therefore, the data synchronization policy must deal with the public interface for integration of data regions.

This raises an interesting question as to the definition of a data atom when the boundaries of the data region are crossed. For the situation where the data atom is identical between the two regions, synchronization is straightforward and any

update of the data atom within a single region is replicated to the other region. Therefore, part of this public interface is the definition of the data atom that needs to be synchronized. There are two approaches to exactly where the data atom definition resides. In the traditional STP/adapter approach, the data representation inherently was addressed through the data load and data store policies. These policies leverage mechanics of the adapters, which know not only how to get data in and out of a region but also the external data interface for the data atom.

An alternative approach would be to define a public interface or method for data atom definition and have it either rolled into the synchronization policy or consumed not by a policy of data management but rather by a query or data access method for the data grid. The latter suggests that the definition of a data atom (for interregion synchronization) is defined externally to the data management policies of the data grid. It is defined as part of the “Service” the grid or data grid supports. In the case of Web Services, XML is the standard method for defining a data atom. Thus, if the data load and data store policies bind the physical adapter to the data movement process, it is that adapter that “understands” the Web Service definition of the data that are being moved. Therefore, data atom definition for interregion synchronization is the responsibility of not the data region but rather the “Service” that the data grid supports and the responsibility of the adapter to manage the mechanics of data translation as part of the data movement process.