Flecc: A Flexible Cache Coherence Protocol for Dynamic Component-Based Systems

Anca Ivan and Vijay Karamcheti
Computer Science Department
Courant Institute of Mathematical Sciences
New York University, New York, NY 10003
ivan,vijayk@cs.nyu.edu

Abstract

An increasing number of distributed applications are currently being constructed as sets of connected components and dynamically deployed in wide area networks using frameworks such as CORBA, .NET, and Web Services. Such dynamic deployments enable applications to flexibly adapt to changes in client QoS requirements and network properties, but introduce a consistency problem because of replicated components. Ideally, the frameworks deploying the applications should ensure that the application consistency requirements are satisfied, even though the requirements can range from weak to strong and dynamically change at run-time. Thus, a key challenge is to design a flexible cache coherence protocol that uses application-specific information while still being application-neutral.

This paper describes Flecc, an application-neutral cache coherence protocol used by a component-based framework (Partitionable Services Framework) to satisfy the consistency requirements of deployed applications. Flecc allows applications to specify appropriate consistency and granularity levels and define complex synchronization decisions as simple functions. We demonstrate the benefits of our cache coherence protocol by analyzing the behavior of a component-based application modeling an airline reservation system.

1. Introduction

Increasingly, distributed applications are being described using various component-based models and deployed in wide-area networks as sets of connected components. Component-based frameworks such as CORBA [19], Globus [6], DCOM [24], the Web Services infrastructure [8], or DCE [13] allow applications to leverage a common substrate that provides essential functionality – e.g. discovery, security, resource management.

Traditionally, such frameworks have relied on static connections between components. However, more recent projects (Active Frames [15], Ninja [21], Active Streams [2], CANS [7], Partitionable Services Framework [9], Conductor [26], and even a more recent version of Globus [6]) have started to advocate the use of dynamically configured component linkages. In such systems, the components are linked at run-time, based on the state of the environment, the client's QoS requirements, and the properties of the application. This dynamic model allows applications to flexibly and dynamically adapt to changes in client's QoS requirements and environment state, possibly by deploying multiple replicas of the same component in the network.

In situations where several replicas sharing data are running in the network, the component-based frameworks should ensure that the application consistency requirements are satisfied. Ideally, applications should provide only the specific functionality and not address additional concerns, e.g. security and data consistency. What makes the cache coherence problem interesting in this context is that it poses different challenges when compared to the same problem in distributed databases, distributed file-systems, or distributed-shared memory systems. In these latter systems, the cache coherence protocols improve their efficiency by making assumptions about the behavior of all applications. In

IEEE Computer Society
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

distributed databases and file systems, clients execute read/write operations at the level of database records, respectively files. In such cases, the caching protocol can take advantage of the data locality and structure (e.g. file systems are organized as trees of files). Similarly, in distributed shared memory systems, the caching protocol uses knowledge about the application implementation (e.g. use patterns for variables) to improve its efficiency. Component-based frameworks deploy general applications and cannot make similar assumptions about either the read/write patterns or data structures that are valid across all applications. In addition, applications deployed in component-based frameworks dynamically adapt to environment and client QoS changes, thus potentially modifying the application consistency requirements. For example, an airline reservation system might allow users to browse flights, buy tickets, and switch between the two modes of operation. In general, users accept stale data during browsing (weak consistency), but require most current data when buying tickets (strong consistency).

Our goal is to design a cache coherence protocol able to work with a wide range of applications. Since it is well known that a caching protocol is more efficient if it uses application-specific knowledge, the main challenge is to permit the use of *application-specific information* while still being *flexible* and *application neutral*. This paper describes the problems found and the solutions developed when designing such an application-neutral cache coherence protocol (Flecc), as part of the Partitionable Services Framework (PSF).

PSF is a component-based framework that deploys dynamic component-based applications in resource constrained environments by flexibly composing the application components. In order to satisfy the consistency requirements of general component-based applications, PSF and Flecc allow applications to specify and dynamically modify the appropriate consistency levels, the granularity levels, and complex synchronization decisions. The application-specific information consists of (i) *data properties*, (ii) *quality triggers*, and (iii) *extract/merge methods*. The properties characterize the data that needs to be kept consistent between replicas. Based on data properties, Flecc decides which replicas share the same data and need to receive updates. The quality triggers indicate when updates should be pushed or pulled between replicas. The extract/merge methods solve the conflict detection and resolution problems by allowing merging and extracting of updates from/into replicas. Even though it uses application-specific information, Flecc is application-neutral because it does not attach semantics to the provided information (e.g. Flecc evaluates the triggers without understanding the semantics associated with the variables).

The rest of the paper is organized as follows. Sections 2 and 3 discuss related work and briefly describe the Partitionable Services Framework. Section 4 gives a detailed description of the cache coherence protocol. Section 5 evaluates the ease of use and the efficiency of our coherence protocol by analyzing the behavior of an application modeling an airline reservation system. We conclude in Section 6.

2. Related Work

The cache coherence problem has been extensively researched in *distributed databases*, *distributed file systems (DFS)*, and *distributed shared memory systems (DSM) for both symmetric multi-processors (SMP)* and *wide-area environments*. The next paragraphs highlight the ways these systems use application-specific information to efficiently satisfy application consistency requirements.

Distributed shared memory - SMPs and cluster environments. Munin [3] and View Caching [12] are examples of software DSM systems that parse application-specific information to choose the appropriate level of consistency at the granularity of pages and objects, respectively. Munin [3] annotates variables with their expected access pattern. The View Caching [12] system defines a view as the data used by a user-defined method and uses view-specific knowledge (data access patterns) to choose the appropriate coherence protocol.

Distributed shared memory - Wide area environments. The problem in DSM systems deployed in WANs is to ensure data consistency between replicas spread across a long latency network, while minimizing the synchronization traffic. In InterWeave [4], applications define the consistency unit as a data segment formed by data blocks and views as subsets of blocks. Views reduce the synchronization traffic, because sharers of the same segment can have different views. Object Views [14] uses a combined run-time and compiler solution to decide which object parts need to be updated for correct execution.

Distributed databases . The consistency problem in distributed databases is to maintain data correctness (e.g. mutual consistency) and availability when multiple read/write operations are simultaneously executed

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

on several replicas. The provided consistency guarantees range from one-copy serializability [1] to weak consistency [5] and continuously weak consistency [27]). In most systems, the information used to improve the consistency protocol efficiency consists of data structures and access patterns [27].

Distributed file systems. Similar to distributed databases, distributed file systems spread information across wide-area networks. Their goal is to allow users to transparently access both their local and remote files [20]. DFSs reduce the synchronization traffic by using information implicit in the hierarchical file system structures when choosing the appropriate granularity levels (pages [25, 18], files [22, 17], volumes [16, 11, 23]).

The common theme across all of these systems is that the underlying cache coherence protocols are able to make assumptions valid across their target application domains and efficiently use this information to design appropriate consistency protocols and define granularity levels.

3. Cache Coherence Problem in PSF

3.1. Partitionable Services Framework (PSF)

Partitionable Services Framework [9, 10] is a dynamic component-based framework which enables construction and deployment of applications as a set of components. PSF allows applications to flexibly adapt to any changes in the client QoS requirements or the network state. The adaptation process consists of assembling and deploying application components into the network, depending on the environment conditions. For example, the security requirements of security-sensitive e-mail application can be satisfied by placing encryption/decryption components around insecure links. Similarly, a cache component placed close to a client can offset high latency of slow links.

PSF relies on four elements: (i) a *declarative specification* of the application and the environment, (ii) *a monitoring module* to follow any changes in the state of the network, (iii) *a planning module* to assemble the components, and (iv) a *deployment* infrastructure to inject the component functionality into the network.

Similar to the CORBA Component Model [19], PSF models components as entities that *implement* and *require* interfaces, where each interface can be associated with properties. The implemented interfaces describe the functionality of the component. The required interfaces indicate what other services are necessary for correct execution of the component. The environment is defined as a set of nodes and links associated with their own properties. The *monitoring module* is responsible for tracking any changes in the state of the environment (e.g. client, network) and trigger adaptation. The *planning module* uses the information provided by the *monitoring module* to find a valid component deployment that satisfies both the application conditions and the client QoS requirements. Once such a composition is found, the *deployment module* securely installs and connects the components in the network.

3.2. Consistency problem in PSF

One of the goals of PSF is to enable flexible access control to the functionality provided by components. Depending on their credentials, users should be allowed to remotely access the components, run components on their local machine, or access the components as a combination of both remote and local execution. In order to solve this problem, PSF defines the notion of *PSF views* [10]. Informally, a view can be (1) a proxy that facilitates the remote access to an original component, (2) a customization of the original component that can be safely executed on the user's local machine, or (3) a new component that allows users to locally execute parts of component and remotely access other parts.

Let's assume that $\mathcal{C}$ is the set of components belonging to an application, and a component $c \in \mathcal{C}$ implements a set of methods F_c and uses a set of variables V_c. A new component $v \in \mathcal{C}$ is a *view* of an original component c if the view has at least one of the following two properties: (i) the functionality of the view is derived from the functionality of the component, i.e. $F_v \cap F_c \neq \varnothing$, and (ii) the data used by the view is a subset of the data used by the component, i.e. $V_v \cap V_c \neq \varnothing$.

The cache coherence problem arises when PSF deploys several views of the same component, as illustrated in Figure 1. In such cases, the views and the original component might require that the shared data be synchronized (e.g. the original component can be regarded as a centralized database, while the data views are replicas of that database). Ideally, PSF should be responsible for ensuring that the application data consistency requirements are satisfied.

There are several challenges in providing consistency

IEEE COMPUTER SOCIETY
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

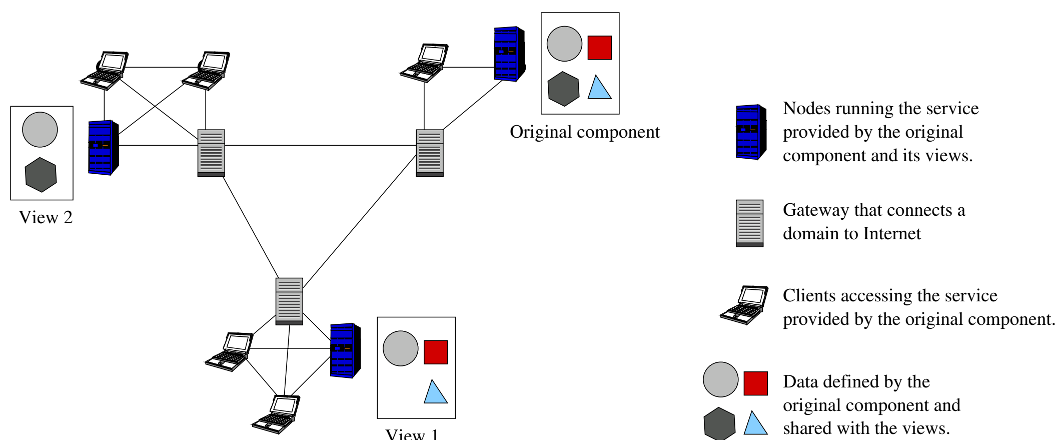

Figure 1. View deployment in PSF. There are three domains connected to Internet. Each domain provides service to its clients through the original component or its views. View 1 and View 2 provide the same service as the original component. Their working data is a subset of the data defined by the original component.

for such component-based applications. First, the cache coherence protocol must work with general component-based applications, without any knowledge on the application internals. Thus, the cache coherence protocol cannot make any assumptions (e.g. read/write patterns or the data structure) that are valid across all applications. The only application-specific information used by the protocol is the one exposed by the application through its interfaces, properties, and requirements. Second, different component-based applications require different consistency levels and the protocol should be able to accommodate all applications. Third, the cache coherence protocol should be able to dynamically adapt to any dynamic changes in the application consistency requirements.

4. Flecc - Cache Coherence Protocol

This paper described Flecc, a cache coherence protocol that satisfies the consistency requirements of any component-based application (*application-neutral*) deployed in any configuration (*flexible*), while using *application-specific information*.

The flexibility of Flecc stems from two factors. First, Flecc acknowledges that different applications have different consistency requirements. Thus, it supports two modes of operation – *strong* and *weak*. The former ensures that there is only active view running in the system, providing essentially one-copy serializability semantics. The latter allows multiple active views to simultaneously work on the shared data and specify more relaxed consistency levels. Second, Flecc allows views to either modify at run-time their weak consistency levels or switch between the strong and weak modes of operation.

To the best of our knowledge, the Partitionable Services Framework is the first component-based framework that provides cache coherence guarantees, which have typically been left to the application. The next two sections describe in detail what type of information is expected from the application, and how the cache coherence protocol uses this information to improve its performance.

4.1. Application-specific information

As explained in Section 3, PSF has no knowledge of the application internals, except what is exposed by the application through its interfaces. In addition, PSF cannot make any assumptions (e.g. data structure or usage) that are valid across all applications. Without additional application-specific information about the shared data, the cache coherence protocol can only execute based on worst-case assumptions, such as that all views conflict and the updates should be sent to all views.

To permit more efficient implementations, Flecc al-

IEEE Computer Society
0-7695-2132-0/04/$17.00 (C) 2004 IEEE

lows applications to provide additional information, which extends the standard interface descriptions. This application-specific information consists of (i) *data properties* to characterize the shared data and indicate *which* views need to be synchronized, (ii) *quality triggers* to indicate *when* updates need to be pushed or pulled between views, and (iii) *merge/extract methods* that define *what* information should be synchronized.

Data properties. Both the original component and the views use *properties* to inform the underlying infrastructure of the characteristics of the shared data. A property p is defined as a tuple $(name_p, D_p)$, where $name_p$ is a unique name and D_p represents the property values. D_p can be an interval $D_p = [d_{min}, d_{max}]$ or a set of discrete values $D_p = \{d_1, d_2, ..., d_n\}$.

Flecc uses data properties to determine which views share the same data, whenever such sharing relationships cannot be statically described. Static relationships are specified into a static map. The map is created once, when Flecc is initialized. The map contains a symmetric matrix, where the number of rows and columns equal the number of views. If two views v_i and v_j share data, than the elements (i, j) and (j, i) in the matrix are set to 1. Otherwise, the elements are set to 0.

Sometimes, it is difficult to statically specify the relationship between two views because they can dynamically change the sets of shared data, i.e. change their data properties at run-time. The static matrix indicates such a possibility by setting the cell entry to -1. In such cases, Flecc uses the *dynamic set of data properties*(see Definition 1) to search for views that share data. Flecc considers that two views v_1 and v_2 share data if the two views are defined by two property sets P and Q and the sets have a non-empty intersection. This method is very flexible and can reduce the coherence traffic by not triggering false conflicts.

$$\begin{aligned} dynConfl &: \mathcal{C} \times \mathcal{C} \rightarrow \{0, 1\} \\ dynConfl(v_1, v_2) &= \begin{cases} 0 & \text{, if } P \cap Q = \varnothing \\ 1 & \text{, if } P \cap Q \neq \varnothing \end{cases} \end{aligned} \tag{1}$$

$$P \cap Q = \{r \mid \exists p_i \in P \; and \; \exists q_j \in Q \; s.t. \; p_i \cap q_j = r\} \tag{2}$$

The intersection of two property sets $P = \{p_1, p_2, ...\}$ and $Q = \{q_1, q_2, ...\}$ is defined as the set of intersections between any two properties of P and Q (see Definition 2). We make the assumption that a set of properties does not contain two properties with the same name (e.g. $name_i \neq name_j \forall i, j$). The intersection between two properties $p = (name_p, D_p)$ and $q = (name_q, D_q)$ is not empty if the properties have the same name and the intersection of the value sets is not empty (see Definition 3).

$$p \cap q = \begin{cases} \varnothing & \text{if } name_p \neq name_q \\ (name_p, D_p \cap D_p) & \text{if } name_p = name_q \end{cases} \tag{3}$$

Quality triggers. Often, applications are responsible for deciding when updates should be either pushed or pulled. One of the goals of Flecc is to simplify applications by taking such decisions on behalf of the application. In addition to making explicit calls to push/pull data, PSF allows views to delegate to the system the right to make the synchronization decisions by defining *push/pull/validity triggers*. Push triggers send the current value of data from the view to the original component. Pull triggers indicate when the view needs to update the shared data with the value held by the original component. Validity triggers are executed whenever the view pulls data and they indicate if the data currently held by the original component is "good enough". If it is not "good", Flecc is responsible for getting the most recent data from the other active views and send it to the requesting view.

If $\mathcal{T}$ is a discrete representation of time, a quality trigger for a view v specifies the synchronization moments as a boolean expression of time ($t \in \mathcal{T}$) and view variables $(x_1, x_2, ...)$, where $x_i \in V_v, \forall i$.

$$T_v(t, x_1, x_2, ...) : \mathcal{T} \times {V_v}^* \rightarrow \{true, false\} \tag{4}$$

There are two ways for the cache manager to evaluate the current values of the object variables: (i) the object provides the necessary methods to access the variables, and (ii) the cache manager uses reflection to examine the variables (when the components are defined in languages that support this feature). The current prototype of PSF is working with Java-based applications, so we use the latter mechanism in our design.

Merge/Extract methods. In order to synchronize the state of all active entities (e.g. views and original component), Flecc needs to propagate the updates from views to the original component and vice-versa. The questions are (i) what information to propagate, and (ii) how to detect and resolve conflicts?

Current systems propagate either *logs of modifications* to be replayed by replicas or *modified data* to be merged into replicas. The first solution does not work in PSF, because views represent different layouts of the same component and might not implement the same methods. Thus, a log defined by one view might not

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

be executable on a different view. Flecc implements the second solution. The challenge in this case is how to extract and merge the data from/into views and the original component if Flecc has no knowledge about the data structure and semantics. The solution is to use application-specific functions to extract and merge data. As in Coda [23] and Bayou [5], Flecc uses these functions to detect and resolve possible conflicts.

All of this information (data properties, quality triggers, merge/extract methods) may be specified by the application for each view and original component. This fact influenced our decision to choose a centralized protocol instead of a decentralized one. The latter regards all entities (i.e. original component and its views) as peers and requires application-specific information on how updates are merged and extracted for every pair of peers ($O(n^2)$). The former takes advantage of the pre-established relationship between the involved entities, i.e. all views are logical representations of the same original component. In this configuration, the original component is regarded as a sink (primary-copy) where all updates are propagated. Thus, the application needs to provide only information on how updates are merged/extracted between the views and the original component ($O(n)$). The downside of the centralized protocol is its assumption that the original component is always running in the system. Fail-safe mechanisms can be implemented; however, they are not the focus of this paper.

In the next section we describe how the cache coherence protocol uses the data properties, triggers, and merge/extract methods to guarantee the required consistency between views.

4.2. Flecc runtime components

Flecc associates a *directory manager* with the original component and a *cache manager* with each view. The responsibility of the directory manager is to keep track of which views are running in the system and control which views are allowed to be active (i.e. working on the shared data). The role of the cache managers is to forward to the directory manager any requests made by the views, and execute the commands sent back by the directory manager. Initially, only the directory manager is assumed to be running in the system. Whenever a view is deployed into the system, the cache manager associated with the view is also created and connected to the directory manager. As part of the creation process, the view provides all the necessary information to the cache manager, as discussed in Section 4.1, which forwards it to the directory manager.

Our cache coherence protocol is similar to classic DSM protocols and consists of two finite state machines executed by the directory manager and the cache manager. The novel feature of Flecc is how it improve the efficiency by using application-specific information (i.e. data properties, quality triggers, and merge/extract methods).

Figure 2 is a simple example that illustrates the interactions between the directory manager and the cache managers associated with two views running in the strong mode. Let us assume that the only active entities in the system are the original component C and its views V_1 and V_2. Let us further assume that the property defined by all entities is P. The values associated to P by the three entities are different: $\{x, y\}$ for V_1, $\{x, z\}$ for V_2, and $\{x, y, z\}$ for the original component.

When the view V_1 is deployed, the view creates a cache manager (step 1) that registers with the directory manager (step 2) and asks for the current data (steps 3,4). The directory manager looks for other views sharing data with the requesting view. Currently, there is none and the directory manager extracts the data from the original component and sends it to the view (step 5). The difference is when view V_2 asks for the current data. In this case, the directory manager finds that V_1 is active and conflicts with V_2. The directory manager sends an invalidation request to V_1, stops V_1 from working, and gives the control to V_2 (steps 12,13,14). This ensures that there is only one active view in the system. In order to prevent the cache manager to merge or extract updates while working on it, the view needs to mark the code that processes the data as mutually exclusive (steps 6,7). At the end, the view announces to the cache manager and thus the directory manager about its intention to stop using the data (steps 20,21).

5. Case study

5.1. Airplane reservation system

We use a component-based application modeling an airline reservation system as an example to illustrate the cache coherence protocol and highlight its benefits. The main components are *reservation clients* of different capabilities (*viewers* and *buyers*), a main *flight database* that contains all information about existing flights, and *travel agents* that can be replicated as necessary to as-

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

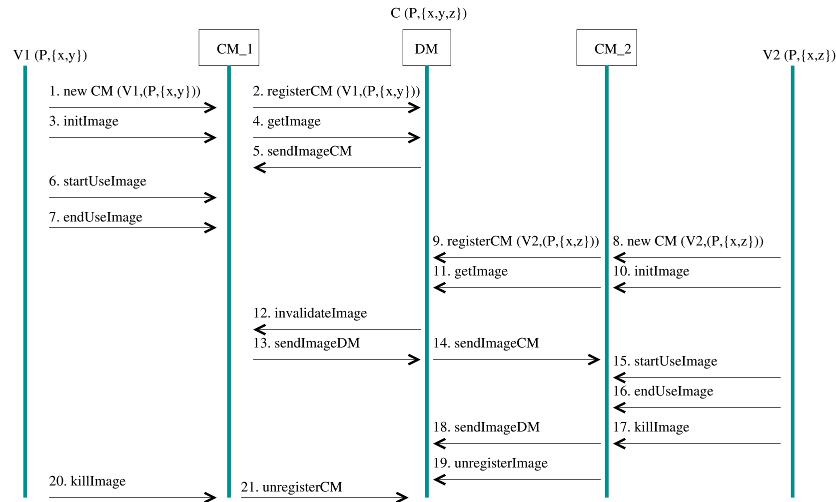

Figure 2. Flecc - Cache coherence protocol

sist the *reservation clients* when browsing the database or buying tickets.

The airline reservation system provides several levels of QoS for clients, where each level is defined by the transaction privacy, the maximum latency for accessing the database, and the type of operations to be performed (e.g. browsing the database or buying the tickets). The privacy of a transaction is ensured by deploying encryptor/decryptor pairs around insecure links. The latency of accessing the database can be decreased by placing travel agent components closer to the clients.

From a caching point of view, the airline reservation system has varied consistency requirements. A viewer does not require the most up-to-date information on flight seat availability. However, a buyer needs fresh information in order to make an educated decision. Thus, the travel agent assisting a view can have more relaxed consistency requirements than a travel agent assisting a buyer. In addition, a viewer can become at any point a buyer and the travel agent component should be able to provide the requested information in a timely manner.

5.2. Evaluation of the cache coherence protocol

We evaluate the benefits of our cache coherence protocol by observing its behavior when PSF deploys the airline reservation application explained in the previous section. We characterize Flecc with respect to whether or not it is *easy to use* (it reduces the number and the complexity of the APIs between the views and the cache managers), *efficient* (it reduces the number of messages sent between cache managers and the directory manager), *adaptable* (it switches between various consistency levels), and *flexible* (allows the application to control the consistency levels by defining quality triggers).

Ease-of-use. As described in Section 4, the API's exposed by the cache manager to the view are natural and convenient to use. Figure 3 illustrates the behavior of the travel agent used throughout our evaluation. The flow of operations is as follows: (1) create cache manager (lines 9-16), (2) initialize data (line 17), (3) work with data (lines 18-29), (4) kill cache manager (line 30). Besides the actual code, the travel agent is also responsible for implementing the extract/merge methods (lines 34-44). Note that this information just communicates what state is extracted/merged and is not concerned with when exactly this functionality is invoked at run-time by the coherence system.

Efficiency. In order to illustrate the efficiency of our cache coherence protocol, we measured the number of messages generated by Flecc and compared it to the number of messages generated by a time-sharing protocol and a multicast-based protocol. The time-sharing protocol allows travel agents to execute one after another. In this way, the number of control messages between the directory manager and the cache managers is kept to a minimum. The multicast-based protocol does not discriminate between cache managers and asks all of them to send updates. Thus, the number of messages between the directory manager and the cache manager reflects the maximum one might see in an application-

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

```
1  public class TravelAgent
2    implements ViewInterface,
3                AirlineReservationInterface {

4   AirlineReservationSystem ars = null;
5   CacheManagerImpl_RMI cm = null;

6   public void run() {
7       ars = new AirlineReservationSystem();
8       cm = new CacheManagerImpl_RMI(
9                    arguments,
10                   "air.TravelAgent",
11                   this,
12                   createPropertyList(),
13                   mode_operation,
14                   "( t > 1500 )",
15                   "( t > 1500 )",
16                   "( t > 1500 )" );
17      cm.initImage();

18      for( int i = 0; i < 10; i ++ ) {
19        cm.pullImage();
20        cm.startUseImage();
21        ars.confirmTickets( 1, flightNumber );
22        cm.endUseImage();
23      }

24      for( int i = 0; i < 10; i ++ ) {
25        cm.pullImage();
26        cm.startUseImage();
27        ars.confirmTickets( 1, flightNumber );
28        cm.endUseImage();
29      }
30      cm.killImage();
31  }

32  /*******************************************/
33  /** IMPLEMENT VIEW INTERFACE **/
34  public static void mergeIntoObject(
35                        Object _obj,
36                        ObjectImage _image,
37                        ViewProp-
ertyList _vpl ) {}

38  public static ObjectImage extractFromObject(
39                        Object _obj,
40                        ViewProp-
ertyList _vpl ) {}

41  public void mergeIntoView( ObjectIm-
age _image,
42                        ViewProp-
ertyList _vpl ) {}

43  public ObjectImage extractFromView(
44                       ViewPropertyList _vpl ) {}
45 }
```

Figure 3. Pseudo-code for a travel agent

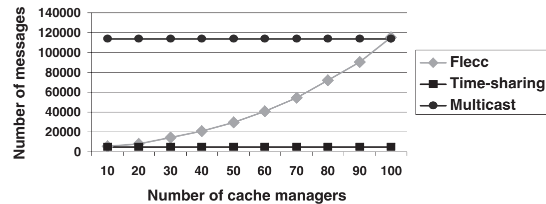

Figure 4. Flecc - Number of messages sent between the cache manager and the directory manager.

oblivious protocol.

The experiment executes 100 travel agent components deployed into a LAN and connected to a main database running in the same LAN. All travel agents execute the same sequence of operations: (1) create the cache manager, (2) set the mode of operation to weak, (3) initialize the data, (4) reserve tickets for a flight, (5) kill the cache manager. Each travel agent defines a property ("Flights") that contains a list of all the served flights. The number of travel agents that serve similar flights is initially 10, and increases in increments of 10 up to 100. The consistency requirements of every travel agent is to always execute on the most current data. Figure 4 shows how Flecc reduces the number of control messages by computing the conflicting travel agents based on their properties. Our cache coherence protocol reduces the number of messages sent between the directory manager and the cache managers, by sending messages only to interested parties.

Adaptability. In order to measure the run-time adaptability of our cache coherence protocol, we use an experiment that deploys ten conflicting travel agents connected to the main database, all running in the same LAN. Initially, they start in weak mode and execute in a loop the "reserve tickets" operation. After that, the travel agents switch to strong mode, and execute the same set of operations. In the last phase, the travel agents switch back to weak and execute the same operations. For this experiment, we measure the time to execute a method and the quality of the data used during the execution. The quality of the data is computed as the number of remote unseen updates to the shared data. Figure 5 shows the trade-off between the time to execute a method and the quality of the data used during the execution. On the Y axis, the graph is split into two parts. The lower part represents the time to exe-

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

cute the methods, while the upper part shows the quality of the data. On the X axis, the graph shows the travel agent execution time line: WEAK, STRONG, WEAK. We observe that the execution time is small when the travel agent is willing to execute on stale data (WEAK mode of operation, where the data quality decreases in time) and increases if the data needs to be most recent (STRONG mode of execution, where the data quality is always the best). More importantly, this trade-off is simply communicated by the application to the underlying system as consistency requirements.

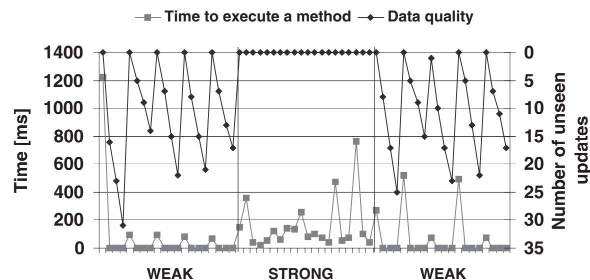

Figure 5. Flecc - Time to execute a method vs. that quality of the used data, when the cache manager switches from WEAK mode to STRONG mode, and back.

Flexibility. We evaluate the impact of the quality triggers on the number of messages and the quality of the data by running ten conflicting travel agents in weak mode, with and without triggers. We measure the quality of the data and the number of messages generated between the cache managers and the directory managers. Figure 6 shows how the quality of data is improved when the travel agents define triggers compared with the case when the travel agent does not specify triggers. The X axis shows the execution time line and marks the moments when the travel agent receives updates. The Y axis shows the measured data quality for every method call. The upper graph represents a travel agent which explicitly pulls the current data before executing four methods. The lower plot represents the same travel agent that uses a time-based pull trigger in addition to explicit calls. However, the cost of the improved data quality is an increased number of messages (116 – no triggers versus 182 – with triggers).

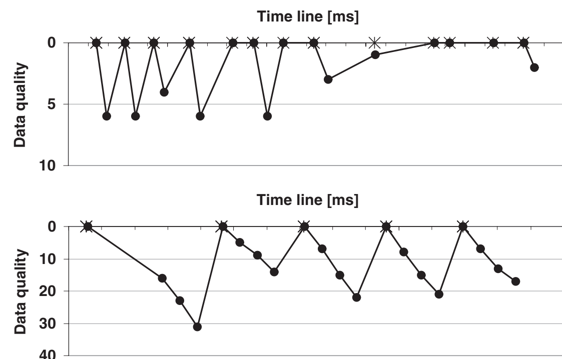

Figure 6. Flecc - Number of remote updates not seen by a cache manager running in WEAK mode. The difference is whether the cache manager defined pull/push trigger or not.

6. Conclusions and Future Work

This paper describes a flexible and application-neutral cache coherence protocol, Flecc, that satisfies the consistency requirements of component-based application deployed in the Partitionable Services Framework. Our cache coherence protocol is capable of adapting at run-time to changes in the consistency requirements. In addition, Flecc achieves efficiency by allowing the application to specify in a natural way with whom, when, and what to synchronize by specifying data properties, pull/push triggers, and extract/merge functions.

There are at least two directions we intend to pursue for improving our cache coherence protocol. First, the cache coherence protocol does not currently use any information about the nature of the methods executed on the shared data. We believe that the number of control messages can be further reduced by attaching read/write semantics to the shared data. Second, the protocol maintains consistency only for the data shared by a single instance of the original component and its views. Flecc could be extended on two levels. The high level protocol would maintain consistency between various instances in a decentralized fashion (e.g. no primary-copy), while the low level protocol would be current version of Flecc and would ensure consistency between components and their views.

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

References

[1] Phil Bernstein and Nathan Goodman. The Failure and Replicated Distributed Databases. In *ACM Transactions on Database Systems*, 1984.

[2] F. Bustamante and K. Schwan. Active Streams: An Approach to Adaptive Distributed Systems. In *HotOS*, 2001.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[4] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. InterWeave: A Middleware System for Distributed Shared State. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.

[5] D. B. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *http://www.globus.org/research/papers.html*, 2002.

[7] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. *USITS*, 2001.

[8] IBM Corporation and Microsoft Corporation. Security in a Web Services World: A Proposed Architecture and Roadmap. *htpp://msdn.microsoft.com/*, 2002.

[9] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogenous Environments. In *HPDC*, 2002.

[10] Anca Andreea Ivan and Vijay Karamcheti. Using Views for Customizing Reusable Components in Component-Based Frameworks. In *12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[11] Thomas W. Page Jr., Richard G Guy., Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus Scalable Replicated File System. Technical Report UCLA-CSD 910005, Los Angeles, CA (USA), 1991.

[12] V. Karamcheti and A. A. Chien. View caching: Efficient software shared memory for dynamic computations. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 483–489, 1997.

[13] Michael M. Kong. DCE: An Environment for Secure Client/Server Computing. volume 46, 1995.

[14] Ilya Lipkind, Igor Pechtchanski, and Vijay Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *OOPSLA*, pages 447 – 460, 1999.

[15] Julio Lopez and David O'Hallaron. Support for Interactive Heavyweight Services. In *HPDC*, 2001.

[16] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, 1986.

[17] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. In *5th Symposium on Operating Systems Design and Implementation*, 2002.

[18] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[19] Object Management Group. CORBA Component Model. *http://www.omg.org/*, 2003.

[20] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[21] S. Gribble at el. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[22] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[23] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[24] W. Rubin et al. *Understanding DCOM*. Prentice Hall, 1999.

[25] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *9th Symposium on Operating System Principles*, pages 49–70, 1983.

[26] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor: A Framework for Distributed Adaptation. In *7th Workshop on Hot Topics in Operating Systems (HotOS VII)*, 1999.

[27] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE