Persistent Objects in the Fleet System

Dahlia Malkhi* Michael K. Reiter† Daniela Tulone† Elisha Ziskind†

Abstract

Fleet is a middleware system implementing a distributed repository for persistent Java objects. Fleet is primarily targeted for supporting highly critical applications: in particular, the objects it stores maintain correct semantics despite the arbitrary failure (including hostile corruption) of a limited number of Fleet servers and, for some object types, of clients allowed to invoke methods on those objects. Fleet is designed to be highly available, dynamically extensible with new object types, and scalable to large numbers of servers and clients. Previous papers described the replication technology underlying Fleet; in this paper we describe the design of Fleet objects, including how new objects are introduced into the system, how they are named, and their default semantics.

1. Introduction

Persistent shared objects are a useful abstraction for building concurrent and distributed programs. A shared object encapsulates state and provides methods to operate on that state that can be invoked from different clients, possibly concurrently and from different physical locations in a network. Persistence of the object implies that the object outlives the client who created it (and other clients); i.e., it will be available to new clients that arrive after its creator has vanished. Implementing persistent shared objects for distributed clients has been the goal of both research efforts (e.g., [LCSA99]) and products alike (e.g., JavaSpaces; see java.sun.com/products/javaspaces/).

Object persistence requires a supporting infrastructure in which object state can be stored. In this paper, we describe Fleet, a software middleware system implementing persistent *Fleet objects* using replication, in a manner that survives the arbitrary corruption of a threshold of a Fleet object's replicas. The Fleet architecture consists of persistent servers and (potentially transient) clients. Servers hold replicas of object state and provide support for implementing method invocations on each Fleet object. Clients share information indirectly through servers by invoking methods on Fleet objects. Previous papers [MR98a, MR98b, MR00, MRW00, MRWW01, CMR01] described the various quorum-based replication protocols supporting object replication in Fleet. These technologies provide Fleet objects with their fault tolerance and scalability features:

- Fault tolerance: Fleet protocols are designed to allow for the arbitrary (Byzantine [LSP82]) failure of a limited number of Fleet servers. For applications that request it, Fleet monitors servers to detect when the number of faulty Fleet servers is approaching that tolerable limit [AMPR99]. The Fleet protocols also tolerate the benign (e.g., crash) failure of arbitrarily many clients invoking methods on Fleet objects. Certain specialized Fleet objects even tolerate Byzantine client failures, but since few objects have useful semantics under Byzantine client failures, Fleet focuses on supporting the benign client case for general objects.
- Scalability: Fleet protocols are designed to scale well in both the number of clients and the number of servers. This is achieved via object replication using *Byzantine quorum systems* [MR98a, MRW00, MRWW01], which enable each operation on a Fleet object to be performed at a small subset of servers at which the object is replicated. This results in better load balancing across servers and lower access costs per operation as compared to prior approaches for implementing Byzantine-fault-tolerant objects, which perform each operation at every server at which the object is replicated.

In this paper, we describe the design of Fleet objects on top of these quorum-based replication technologies. Specifically, here we focus on the following properties provided by Fleet objects:

*School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel; email: dalia@cs.huji.ac.il

†Secure Systems Research Department, Bell Labs, Lucent Technologies, Murray Hill, New Jersey, USA; email: {reiter,daniela, eziskind}@research.bell-labs.com

This document is based upon work supported by DARPA and the U.S. Air Force Research Laboratory under Contract No. F30602-99-C-0165. Any opinions, findings, conclusions or recommendations expressed in this document are those of the author(s) and do not necessarily reflect the views of DARPA or the U.S. Air Force Research Laboratory.

0-7695-1212-7/01 $10.00 © 2001 IEEE

- Concurrent semantics: Fleet provides a generic protocol for turning any regular Java object into a shared object with *linearizable* [HW90] concurrent semantics. Intuitively, this means that the return values from (possibly concurrent) invocations on the object are the same as those returned in some execution in which each operation is executed instantaneously at a distinct point in real time between the time of its invocation and the time of its response. In particular, linearizable Fleet objects can be used in applications as if they are accessed sequentially.

- Liveness: During periods of *stability*—i.e., communication between clients and servers is timely—operations on a Fleet object complete with probability one. More precisely, t concurrent operations on a Fleet object complete in expected time $O(t)$ once the system stabilizes. Certain specialized Fleet objects offer even stronger availability guarantees, such as wait-freedom [Her91]. However, since wait-freedom is known to be impossible to implement for general objects [Her91], conditional liveness properties such as ours are a necessity.

- Autonomy: Each Fleet object is replicated at a set, or *universe*, of Fleet servers designated by the application, depending on the level of fault tolerance desired by the creator of the object. In particular, the universes of different Fleet objects can be chosen independently, and may or may not overlap.

- Isolation: Fleet supports configurations in which multiple users' or applications' Fleet objects are implemented using possibly overlapping universes of Fleet servers. To support this, each Fleet object is created within a *name space* that groups the Fleet objects of a user or application, and that allows only the creator of the name space to place new objects within it. Provided that an application searches for Fleet objects in the proper name spaces, the presence of objects in other name spaces—even if implemented on the same Fleet servers—will not interfere with the application. Name spaces are location-independent, i.e., a name space can contain Fleet objects replicated at entirely distinct universes of Fleet servers.

- Extensibility: Fleet supports the dynamic introduction of new protocols for implementing Fleet objects. In this way, an application developer can use standard Fleet objects where convenient (i.e., Fleet objects using the standard Fleet protocols) and can customize these protocols, or create entirely new protocols, on a per-object basis. Such custom code is dynamically loaded into Fleet and, using Fleet itself for persistent storage, is available for use by other clients and by the Fleet servers.

The main goal of this paper is to provide an overview of the abstractions that Fleet supports for the application developer, though a coherent coverage also requires some introduction to Fleet internals. Particular algorithms and protocols used in the system will not be detailed here. The Fleet architecture, and several of the key Fleet protocols, e.g., for implementing various kinds of linearizable objects, for monitoring the system for faults, and for diffusing updates throughout the system, have been detailed elsewhere (see [MR98b, AMPR99, MMR99, MR00, CMR01]).

The remainder of this paper is structured as follows. In Section 2 we describe related work. We minimally introduce the Fleet architecture on Section 3, and describe the behavior of Fleet objects in Section 4. Naming Fleet objects, and the requisite mechanisms for fully defining a Fleet object, are described in Section 5. We provide some cautions regarding manners of combining Fleet objects in Section 6. In Section 7, we briefly describe the aspects of a Fleet server that are relevant to a developer. We provide preliminary performance results for Fleet in Section 8, and conclude in Section 9.

2. Related work

We begin by placing the concurrent semantics of Fleet objects into perspective. In Fleet, the default object sharing semantics provided is linearizability [HW90] (though other semantics are provided by specific Fleet objects). Linearizability ensures atomic (indivisible) method invocation semantics on replicated objects as defined by Lamport [Lam86]. Generally, there are three data replication techniques supporting linearizable semantics: primary-backup (PB), in which a designated primary server responds to each method invocation; state machine replication (SMR), in which each each server performs and responds to every method invocation; and quorum replication as used in Fleet. PB is not suited to masking Byzantine server failures (since the primary could fail), and so Fleet differs from PB systems in this regard. Though SMR can mask Byzantine failures (e.g., see [Sch90]), Fleet differs in its attention to scaling as a function of the number n of servers over which an object is replicated. Because of its underlying quorum replication technology, clients can perform operations on Fleet objects by interacting with as few as $O(\sqrt{n})$ servers. The price that Fleet pays for this better scalability, however, is weaker fault-tolerance: state machine replication can mask up to $n/3$ faulty servers, whereas the quorum techniques underlying Fleet can mask at most up to $n/4$ [MR98a]. Another differentiating aspect of Fleet is its extensibility, which to our knowledge has not been a goal of prior systems of this type.

Concurrent semantics stronger than linearizability include transactional semantics, whereby a group of operations (a transaction) on multiple objects, possibly nested, is done indivisibly. Transactions are traditionally sought in database systems; for a comparison of replication techniques in databases and distributed systems, see [WPS+00]. Our design differs in its attention to scale, and to arbitrary failures, whereas database replication methods traditionally focus on smaller systems with benign failures only.

Fleet can also be compared to persistent, distributed storage facilities for Java objects, notably JavaSpaces. Objects can be written to and read from a JavaSpace, and support is provided for handling concurrent access and performing distributed transactions. JavaSpaces itself does not address fault tolerance issues, but other systems built on top of it have attempted to address these issues. One such system is Globe [LS99], which extends JavaSpaces to provide support for benign fault tolerance (by using replicated JavaSpaces) and scalability (by partitioning a set of JavaSpaces into disjoint subsets). Fleet's attention to arbitrary failures distinguishes it from systems such as Globe.

Also related to Fleet are several systems implementing scalable data storage using wide-scale replication, such as the Eternity service [And96], Publius [WRC00] and Intermemory [CEG+99]. A first difference from Fleet is that these systems do not support richer objects than read-write files. Moreover, even the read-write files they support are intended to be primarily read-only, as they do not specify semantics for reads that execute concurrently with writes.

Finally, the Fleet effort grew out of a predecessor system called Phalanx [MR98b], which was based on similar principles as Fleet, e.g., quorum access with high availability and tolerance to arbitrary failures. However, Phalanx only implemented read-write files (versus arbitrary objects), was not designed for extensibility, and reached a level of maturity and performance far below our goals in Fleet. Fleet shares only some design aspects with Phalanx; no Phalanx code, in particular, is present in Fleet.

3. Overview of the Fleet architecture

A "Fleet object" is merely an abstraction. There is no single Java object that implements it. Rather, it refers to the collection of client-resident stubs and server-resident object replicas that together provide clients with an emulation of a shared object. The client-side stubs for Fleet objects are called FOHandles (for "Fleet Object Handle"). On the server side, Fleet servers store object replicas called FOReplicas (for "Fleet Object Replicas"), each of which maintains a copy of the state of a Fleet object and supports method invocations on it. Together, FOHandles and FOReplicas implement protocols for executing a client's method invocations on a FOHandle, while hiding from the client the details of performing these accesses consistently across FOReplicas. Figure 1 depicts these Fleet object components residing at servers and clients. FONames (for "Fleet Object Name") are used to uniquely describe a Fleet object; from these, both FOHandles and FOReplicas can be generated.

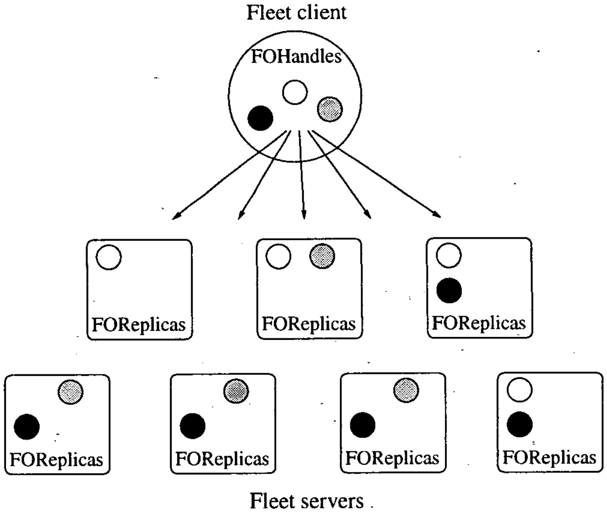

Figure 1. Fleet object components

FOReplicas are accessed using *Byzantine quorum systems* [MR98a, MRW00, MRWW01] for efficiency, survivability and scalability. Briefly, a quorum system is a set of subsets (quorums) of servers (i.e., FOReplicas) such that any two quorums intersect. Fleet objects are constructed with quorum systems designed to mask b arbitrary server failures, which typically requires that quorums intersect in $2b + 1$ or $3b + 1$ FOReplicas depending on the method invocation protocol used to implement the Fleet object (see [MR98a, CMR01]). Our claims of scalability for Fleet derive largely from the use of quorums in these protocols. For example, Byzantine quorum systems can be designed with quorums of size only $O(\sqrt{bn})$ where n is the number of servers (FOReplicas) used in the implementation of the object [MRW00, MRWW01].

FOReplicas are deployed on Fleet servers on an as-needed basis. That is, a FOReplica for a Fleet object is instantiated on a server (from a FOName; see Section 5) only once that server is invoked by a FOHandle for that Fleet object (or otherwise receives updates for it; see Section 7.3). Since only a quorum of servers is involved in emulating a method invocation on a Fleet object, a FOReplica may not be deployed on a particular server in the Fleet object's universe for an indefinite period of time.

4. Fleet objects

A Fleet object looks to an application much like a *remote* Java object. In Java, a remote object is one whose methods can be invoked from outside the Java Virtual Machine (JVM) in which it resides, and the process of doing this is called *remote method invocation* (RMI). Before invoking a method on a remote object, the calling process must obtain a reference (called a *stub*) for the remote object. Each method invocation on this stub is then translated to a method invocation on the remote object.

Java's RMI is a useful analogy for how an application works with Fleet objects. A reference to a Fleet object, i.e., the analog of an RMI stub, is a FOHandle. An application can invoke methods on the FOHandle, and each such invocation emulates an invocation of the "remote" Fleet object to which it refers. Rather than remote invocation on a single remote object, method invocations of a FOHandle are implemented using distributed protocols involving the collection of FOReplicas that reside on Fleet servers.

4.1. Method invocation semantics

The semantics of method invocations on Fleet objects are completely dependent on the implementations of the FOHandle and FOReplica for the Fleet object. There are three ways in which Fleet objects (i.e., FOHandles and FOReplicas) can be constructed in applications.

Off-the-shelf objects. An application may instantiate and use certain types of objects that are implemented in Fleet already, of which there are a few. One type of object supported in Fleet deserving special attention is a *write-once file*, whose method invocation semantics tolerates arbitrary client failures, in addition to a limited number of arbitrary server failures. This Fleet object enables its creator to write a Java object (of an arbitrary serializable type) to this file, but once this is done, the Fleet object cannot be modified any more. Moreover, if two writes to the file are attempted concurrently, at most one (but possibly none) will succeed. Therefore, if any two clients read this file, and both reads return an object, then the returned objects are the same. In this sense, this Fleet object implements an abstraction much like Byzantine agreement [LSP82]: no two clients can read different objects from this file, even if the writer of the file does not follow the correct write protocol. It is strictly weaker than Byzantine agreement, and hence possible to implement, since it may remain unwritten forever if no client succeeds in writing it. This Fleet object is very useful within the system, as will be discussed in Section 5.3. The protocol by which this object is implemented is similar to the "untrusted writer" protocol of [MR98b].

Custom objects. A sophisticated application developer can implement a custom object by developing her own FOHandle and FOReplica, to execute a method invocation protocol enforcing any method semantics she chooses.[1] For example, the programmer has flexibility in terms of the semantics of concurrent method invocations on the Fleet object, and the behavior of the method invocations, in the face of benign or arbitrary failures of clients (i.e., FOHandles) or Fleet servers (i.e., FOReplicas). The steps an application developer must take in order to introduce custom FOHandles and FOReplicas into Fleet are discussed in Section 5.3.

Default method invocation protocol. Because distributed protocols for tolerating failures (particularly of an arbitrary nature) and for ensuring correct semantics in the face of concurrency can be quite complex, we expect that most applications will not build custom method invocation protocols for their Fleet objects. As a result, in Fleet we have focused on providing support for what we expect to be the method invocation semantics desired by most applications. Specifically, Fleet provides a method invocation protocol [CMR01] whose concurrent semantics mimic those of a Java remote object: each executes at some time between its invocation by the application and its response to the application, and even concurrent invocations appear to execute in some serial order. Formally, this is known as *linearizable* semantics [HW90]. An application developer can utilize this protocol to "Fleet-ify" any object and provide distributed emulation of the object that guarantees serialization of its method invocations consistently on replicated servers. For example, an application may have a Java object implementing the interface of a FIFO queue (with methods "enqueue" and "dequeue") and turn it into a Fleet object with the same interface using the default method invocation protocol. Fleet provides tools for automatically generating Fleet objects in this way.

The default method invocation protocol tolerates a number of arbitrary server (i.e., FOReplica) failures that is configured when the Fleet object is created, and any number of benign client failures. We opted to support only benign client failures in this protocol because few objects provide useful semantics if accessed by arbitrarily faulty clients—e.g., a file object that can be overwritten by a client that fails arbitrarily (and thus can write garbage to the file) can be tolerated by few applications—and because protocols for accommodating arbitrary client failures are much more expensive (c.f., [MR00, MMRT00]).

[1] The programmer is, however, limited by security restrictions on FOHandles and FOReplicas, as will be discussed in Subsection 7.2.

4.2. Liveness

The liveness semantics of Fleet objects refers to the guarantees provided regarding when method invocations on Fleet objects will return. In general, these depend on the implementation of the method invocation protocol.

Fleet's default method invocation protocol probabilistically ensures completion of each method invocation during periods of system *stability*, i.e., when message transmission delay between Fleet clients and servers is bounded by a known constant. More specifically, when the system is stable, this protocol ensures that t concurrent method invocations on the Fleet object complete in *expected* $O(t)$ time.

Particular Fleet objects (e.g., see [MR98b]) provide a stronger liveness property called *wait-freedom* [Her91]. Informally, this means that any method invocation by a client that does not fail will eventually complete and return. In Fleet, this property is conditioned on an assumption about the number of servers that simultaneously fail but is *not* conditioned on the behavior or failure of other clients. Indeed, independence from other clients is intrinsic to the definition of wait-freedom. Unfortunately, wait-freedom is known to be impossible to implement for any object with semantics stronger than a read-write file [Her91]. As a result, our richer Fleet objects provide necessarily weaker liveness guarantees.

5. Fleet object naming

To create a FOHandle for a Fleet object, an application must first obtain a FOName ("Fleet Object Name") for the Fleet object. A FOName is simply a collection of fields that together constitute a globally unique identifier for the Fleet object. In this section we introduce the fields that constitute a FOName and explain their significance for the implementation of the corresponding Fleet object.

Specifying these fields necessarily exposes some internals of the Fleet architecture. As a result, an application programmer must be familiar with these internals in order to create and deploy new Fleet objects in her application. However, these details can be hidden from an application that uses only already-deployed Fleet objects; simply retrieving opaque FONames for the objects of interest suffices to enable an application to invoke methods on them. In particular, Fleet provides a utility class that takes a FOName and returns an appropriate FOHandle for invoking the Fleet object with that FOName.

5.1. Quorum system

As discussed in Section 3, the Fleet protocols are designed around the principle that a FOHandle need communicate with only a *quorum* of FOReplicas to complete a method invocation on the Fleet object it represents. The application is free to choose for each Fleet object the Fleet servers at which the FOReplicas will reside (the "autonomy" property in Section 1), the number of arbitrary server (FOReplica) failures the Fleet object will tolerate, and the quorums used for accessing the object. Hence, a component of a FOName is a quorum system description, including a list of Fleet servers at which FOReplicas should exist for the Fleet object and the subsets of them that constitute quorums. A quorum system description must be provided by the application that creates the Fleet object.

5.2. Name space

Each Fleet object is created within a *name space* specified by the application. Abstractly, a name space groups Fleet objects that are somehow related, e.g., as part of the same application or created by the same user. A name space provides a primitive form of access control, in that generally only the application that creates a name space may create Fleet objects within it or delete Fleet objects from it. It is this mechanism that provides the "isolation" property described in Section 1.

A name space is identified within a FOName by a public key of a digital signature scheme, presently either an RSA [RSA78] or DSA [Kra93] public key. An application can create a new name space at any time simply by generating a new public key pair locally. Any Fleet object whose FOName contains that public key is then considered part of that name space, provided that the FOName also includes a digital signature of all other fields that can be verified using the public key that it contains.

The requirement that each FOName contain a valid digital signature by the private key corresponding to the public key of its name space enforces the aforementioned access control. This access control implements a useful form of isolation between different applications, since it prevents the mistaken or deliberate insertion of Fleet objects into name spaces where they do not belong. That is, if a distributed application employs only Fleet objects from a certain name space whose creator is trusted, then the creation of new Fleet objects by others (necessarily in a different name space), even with FOReplicas on the same Fleet servers, will not interfere with the application. Similarly, Fleet servers require a request signed by the private key corresponding to the public key in a FOName before deleting the local FOReplica for the Fleet object with that FOName. So, others cannot capriciously delete an application's Fleet objects.

These isolation properties may be restrictive for some applications, particularly those in which, e.g., the parties creating and deleting a Fleet object are intended to be different. More flexible access control mechanisms are needed for such cases. As a simple example, presently Fleet enables a Fleet object to be created within a name space that is *subordinated* to that of the party intended to delete it. Fleet implements this by creating a new "subordinate" name space (i.e., public key pair) and specifying this new name space in the FOName of the new Fleet object. Fleet then adds an additional field to this FOName, which is the private key of the subordinate name space encrypted under the public key of the deleting party's name space. In this way, the deleting party can recover the private key of the subordinate name space to delete the Fleet object (or similarly add new objects to the name space).

5.3. Code files and initialization arguments

As discussed previously, an application can dynamically introduce Fleet objects and, more specifically, a new FOHandle and FOReplica implementing a new type of Fleet object. For such a Fleet object, it will be necessary for clients other than the Fleet object's creator to obtain the Java class files (i.e., executable code) and initialization arguments for the FOHandle of this Fleet object in order to instantiate one. Similarly, a Fleet server will need to obtain the class files and initialization arguments for its FOReplica in order to instantiate and host it.

Java itself provides support for automatically disseminating class files for Java objects. When an object is serialized, a URL naming a web server where the class file for the object is located can be embedded in the serialized version of the object. In this way, when the object is deserialized, the Java runtime can retrieve the class file from this URL and reconstitute the object. We have opted against using this facility in Fleet, however, for several reasons. First, if a client or Fleet server relied on a web server to retrieve the class files for a Fleet object, then this Fleet object would be only as available as that web server. Second, this mechanism is unnecessary for Fleet, since Fleet already has available a much more reliable means of disseminating information such as class files, namely Fleet itself.

More specifically, a FOName for each Fleet object O contains a field that is a FOName of a write-once file (itself a Fleet object) as described in Section 4. This write-once file, called the *code file* for O, is intended to contain all of the Java class files (in the form of a Jar file) needed to instantiate and execute a FOHandle or FOReplica for O.

The application that creates O is required to provide the FOName of a code file that can be embedded in the FOName of O. For example, an application can create the code file using the same quorum system as is provided for O, so that the code file has at least the fault-tolerance and availability characteristics as O. By this indirect reference to the code file, the FOName for O is kept reasonably sized, and furthermore, a single code file's FOName can be listed within several different Fleet objects' FONames, thereby avoiding having many copies of the class files. Note that the FOName for the code file itself contains a null FOName for its code file, since the class files needed to implement a write-once file are part of the core Fleet distribution; i.e., any Fleet client or server should already have them.

Any party holding the FOName for O can then extract the FOName of its code file, instantiate a FOHandle for it, read the Jar file it contains, and then instantiate a FOHandle or FOReplica for O. This entire process is performed invisibly to the application and is integrated into Java's ClassLoader mechanism so that only if a class file is not available locally is a code file consulted. Moreover, since there is a separate ClassLoader per code file, name clashes between the class names of different applications are avoided.

An application writing a code file needs to determine all the relevant class files and collect them into a Jar file. This task is assisted in Fleet by a dynamic loading facility that infers for any given Java class (or package) the additional classes required to execute it. The dynamic loading facility scans the byte-code of its parameter class(es) to find any dependencies on other classes. The reason for performing a byte-code scan is that Java's *class reflection* does not expose fields used internally by methods, which are only reflected in the byte-code.

Initialization arguments for the FOHandle and FOReplica of a Fleet object could similarly be provided within a write-once file. However, presently in Fleet we explicitly list these arguments within the FOName of the Fleet object. We have chosen this because we expect initialization arguments to be smaller and possibly more variable among Fleet objects of the same type.

5.4. String description

The quorum system, name space, class files and initialization arguments made accessible in a FOName enable any party to create a FOHandle or FOReplica for the corresponding Fleet object. They also enable a server to determine whether the creator of the Fleet object intended for one of its FOReplicas to be hosted at that server. However, these fields are not enough to give each Fleet object a unique name, since it is our intention that applications be able to create multiple Fleet objects of the same type (and possibly initialization arguments) using the same quorum system and name space. For this reason, there is an additional field in a FOName that an application can specify, namely a string description of the Fleet object. This is included in the FOName mainly so that applications can provide different descriptions and/or text names for different Fleet objects. An application can additionally use them in any way it chooses, e.g., to provide a human-readable description of the object's purpose and semantics.

6. Combining Fleet objects

It may be natural in applications for one Fleet object to refer to other Fleet objects. For example, an application may have a Java object implementing the interface of a FIFO queue (with methods "enqueue" and "dequeue") and turn it into a shared queue O using the default method invocation protocol. It then may need to enqueue other Fleet objects on O. There are two potential pitfalls that the application developer should consider when combining Fleet objects in this way.

First, care must be taken to ensure that another client that dequeues a Fleet object from O can instantiate a FOHandle for it, in case the dequeued object is one for which the class files are not already stored locally at that client. This can dictate how Fleet objects must be enqueued on O, for which there are two possibilities:

1. In the first approach, FOHandles of Fleet objects are enqueued on O. This approach is the easiest to integrate with existing applications—as when replacing local or remote Java objects with Fleet objects in an existing application—since the FOHandle simply takes the place of the object or stub being replaced. However, this approach is guaranteed to work only if the class files for any Fleet object enqueued on O are also available in the code file for O (see Section 5.3). If instead a client instantiates a FOHandle for O and dequeues an element from it, and if this element is a FOHandle for a Fleet object for which the client does not possess the class files,[2] then its attempt to instantiate the element will result in a ClassNotFoundException. Thus, this approach is best employed when both O and the Fleet objects enqueued on it are part of a single application for which the class files can be written to a single code file.

2. In the second approach, FONames of Fleet objects are enqueued on O. This approach is somewhat more complex to integrate with existing applications, since replacing one object of the application with a Fleet object may require its FOHandle in some places (where the object is invoked) and its FOName in others (where the object is enqueued on O). However, this approach does not require any prior arrangement of the class files for O and other Fleet objects enqueued on it; rather, they can be treated completely independently.

A second potential pitfall arises from the fact that Fleet does not support transactional semantics. Thus, if a method invocation on a Fleet object results in a nested call on another Fleet object, no guarantee is made on the semantics of this multi-object operation. Indeed, the present implementation would risk performing the nested invocations of Fleet objects multiple times, and in arbitrary orders relative to other invocations on those objects. In this sense, Fleet objects do not *compose*. Stronger semantics for nested invocations of Fleet objects is a topic of ongoing work.

7. Fleet servers

So far we have said little about Fleet servers, since we have focused in this paper on the abstraction of Fleet objects provided to applications. Nevertheless, certain aspects of Fleet servers are relevant to application developers who build custom Fleet objects, and so in this section we discuss these features.

To a first approximation, a Fleet server is a hollow shell in which FOReplicas can execute: the Fleet server does little more than dispatch each incoming request to a FOReplica (possibly after creating the FOReplica as discussed in Section 5.3) and return the FOReplica's response, after checkpointing the FOReplica state to disk if necessary. In particular, the actual method invocation protocols for Fleet objects are implemented in the FOReplicas themselves. Thus, the behaviors of Fleet servers that are mainly relevant for application developers are in how they make themselves and the Fleet objects they represent accessible to clients, and in how they restrict the behavior of FOReplicas they host for security purposes.

7.1. Advertising Fleet objects

A Fleet server provides an interface that enables any client to learn the FONames of Fleet objects for which the server currently is hosting a FOReplica. This interface provides rich searching capabilities to enable clients, e.g., to search for FONames in particular name spaces or with particular string descriptions, or representing Fleet objects of a particular type. As described in Section 5, once a client obtains the FOName for a Fleet object, it can instantiate a FOHandle for it and invoke methods on it.

Therefore, the remaining challenge for a Fleet client is to find Fleet servers themselves. This is not a problem unique to Fleet servers, in that any network service must make some means of addressing it available to potential clients. We anticipate that most applications needing the reliability and security of Fleet would also predistribute addresses of their intended (and presumably trusted) Fleet servers to its client components. Nevertheless, Fleet servers can further employ any means of advertising their presence that is useful for potential clients. At the time of this writing, for example, each Fleet server advertises itself as a Jini service

[2] More precisely, the same ClassLoader that loaded the FOHandle for O must have access to these class files. Since a separate ClassLoader is instantiated per code file, this implies that these class files must be available in the code file for O.

(see `www.sun.com/jini/`) that can be located via the Jini discovery protocol. We emphasize, however, that this is merely for convenience; Jini is not sufficiently secure or reliable to act as a basis for critical applications to find their intended Fleet servers.

7.2. Restricting FOReplicas

Since FOReplicas for Fleet objects of potentially mutually distrusting applications can coexist on the same Fleet servers, it is important that a Fleet server isolate FOReplicas from one another. For example, if one FOReplica could corrupt another on the same Fleet server, then the failure assumptions on which the latter's Fleet object are based could be at risk of being violated. Because FOReplicas execute in a Fleet server within the same JVM, the problem of isolating one FOReplica from another is similar to the problem of isolating one Java applet from another in the same JVM. A solution to the latter has been a central requirement underlying Java's viability as a mobile code technology, and correspondingly has received a considerable amount of attention (see, e.g., [MF99]). We are fortunately able to bring much of this development to bear on the problem of isolating FOReplicas from one another.

Direct interaction between FOReplicas in the same Fleet server is limited by disallowing one FOReplica to obtain a reference to another. Type safety of Java then prevents one FOReplica from invoking the methods of another, for example. In particular, FOReplicas are *not* remote objects, and so methods of a FOReplica cannot be directly invoked from outside the Fleet server. Limiting the propagation of references to FOReplicas within the JVM, and making judicious use of Java's access levels (i.e., private, protected, public) for methods and variables of FOReplicas, help to ensure that FOReplicas can be invoked only by the Fleet server classes (versus other FOReplicas) and in the intended ways.

Indirect interactions between FOReplicas is a broader issue that we address mainly by limiting the capabilities granted to each FOReplica. FOReplicas are "sandboxed" with even fewer default privileges than would be granted to an untrusted applet loaded into the JVM of an off-the-shelf browser. In particular, FOReplicas do not have privileges to conduct network I/O or disk I/O, and thus cannot manipulate the checkpoint files of other FOReplicas, for example.[3]

One threat that we have not fully addressed is resource clogging attacks whereby, for example, a FOReplica could create an enormous object of size larger than the maximum heap size for the Fleet server and thus cause the Fleet server to halt. Approaches to restrict, or monitor and react to, resource usage of untrusted code running within a JVM is the topic of much current and promising research (e.g., see [HCC+98, TL98, CvE98, BHL00, RCW01]). We anticipate that such work will soon yield better JVMs that offer both the performance advantages of running within a single JVM and less vulnerability to resource-consumptive code. Fleet servers (and clients) should make use of such platforms when they become available.

7.3. Propagation

One service that Fleet servers provide for FOReplicas is *update propagation*, by which an update applied at one FOReplica can be diffused to the other FOReplicas of that Fleet object. Update propagation is done in the background, outside any specific Fleet object protocol, and transparently to the client application. While the properties of Fleet objects that we have described so far do not require propagation, propagation can nevertheless be useful to improve a Fleet object's properties when certain quorum systems are used or in the case of a network partition; see [MMR99].

A Fleet server must be informed about which updates to propagate to other servers. To this end, each FOReplica can export an event whenever it is updated. This event encapsulates the object-specific update to be propagated. The propagation module in a Fleet server monitors these events for every FOReplica it hosts. It maintains the updates captured in these events in a set to propagate to other Fleet servers.

Periodically, each server propagates a collection of recently accumulated updates to other servers. The algorithm determining the target of diffusion is described in [MMR99]. The implementation aggregates diffusion for multiple Fleet objects and sends a collection of updates in one message in each round of propagation. This is complicated, however, by the fact that different Fleet objects have FOReplicas at different sets of servers. Therefore, in each round, a server selects targets for all of its of pending updates, such that there is one target for each update. It then packs all updates destined to a particular target in one message, and sends it. In this way, the cost of diffusion is amortized over multiple object updates, to the extent possible by their respective universe overlap.

When a server receives a collection of propagated updates, it sorts them into "buckets" of identical updates per Fleet object. Only when an update is received from $b + 1$ different servers hosting a FOReplica for this Fleet object, where b is the resilience threshold of the object's masking quorum system, it is accepted and applied to the local FOReplica for that Fleet object. The delay analysis in [MMR99] indicates how many propagation rounds it will take an update to reach the entire system. The Fleet servers garbage collect the updates from their propagation queues based on this predicted delay.

[3] Checkpointing does not require that FOReplicas have disk access permissions. Rather, the Fleet server performs the disk I/O needed to checkpoint the state of a FOReplica.

8. Performance

In this section we report performance results for our present implementation of method invocations on a Fleet object using Fleet's default method invocation protocol. We describe performance for two variations of this protocol, which we refer to here as "general" and "deterministic". These terms refer not to the implementation of the protocol, but rather to the kinds of Fleet objects they support: the "general" protocol supports even Fleet objects with nondeterministic interface specifications, whereas the "deterministic" protocol requires that the Fleet object have a deterministic specification. In terms of performance, the primary distinction between these protocols is that the general protocol invokes methods on the state of a Fleet object at a client (i.e., the client resolves any nondeterminism) and copies the object state back to a quorum. In the deterministic protocol, method invocations on the state of a Fleet object are performed by each server in a quorum.

The cost of the generality in the general protocol comes primarily as the state of the replicated Fleet object grows, since the full state is copied back to servers. For this reason, in this section we execute tests involving three types of objects, namely "small", "medium" and "large". A "small" object has a state of negligible size. A "medium" object has a state of size roughly 158 Kbytes in these tests. A "large" object has a state of roughly 312 Kbytes.

Other parameters of these tests are the quorum size q, the number n of FOReplicas of the Fleet object being invoked, and its fault-tolerance threshold b. To provide a conservative estimate of performance, we chose relationships among these parameters to induce worst-case performance as a function of the quorum size q. In particular, we chose a variation of the *threshold* quorum system of [MR98a] that exhibits relatively poor load-balancing properties (but is otherwise attractive for its fault-tolerance and simplicity); in this quorum system, $q = \lceil (n + 3b + 1)/2 \rceil$. In addition, b was chosen to be the maximum possible, i.e., $b = \lceil n/5 \rceil - 1$. This choice of b minimizes n for a given q, thereby further limiting opportunities for load balancing in tests involving multiple clients, and maximizes the processing performed by each client (which grows as a function of b).

The tests reported in this section were performed on a collection of 450 MHz Pentium II computers running RedHat Linux 6.1 and JDK 1.3 with HotSpot. The number of computers used in our tests was limited to 10. Thus, for any $q > 6$, at least one computer was required to host multiple Java virtual machines running either a FOReplica or a client's FOHandle. This resulted in contention for the computer's resources when the protocol execution required participation of multiple components on the same host. This is one more way in which the numbers of this section present a conservative picture of Fleet performance.

The average method invocation response time experienced by a single client that repeatedly invoked a method on a Fleet object is shown in Figure 2. The curves in this graph demonstrate performance for various combinations of small, medium or large Fleet objects and the general or deterministic protocol. The performance of the general and deterministic protocols are the same for small objects, and so there is only one curve for both of these cases. Each plotted point is the average response time as computed over 150 invocations. As discussed previously, the performance difference between the general and deterministic protocols for medium and large objects results from technical differences between these protocols that results in more generality for the first but also more overhead when used with Fleet objects with nontrivial state.

Average method invocation response time experienced by each of multiple clients invoking methods concurrently on the same Fleet object with quorums of size $q = 10$ is shown in Figure 3. In these tests each client performed 200 method invocations sequentially on the object. All clients started at roughly the same time (upon receiving a multicast signal to begin). The plotted points are the method response times, obtained by first averaging each client's 200 method invocation response times and second averaging these values across all clients. These averages are plotted versus the number of clients.

Our belief that method invocation response time will scale well as a function of n derives from the observation that for an appropriate choice of quorum system—of which our choice above is *not* an example—n can grow roughly quadratically as a function of q (see [MRW00, MRWW01]). Therefore, it is reasonable to expect that the performance demonstrated in Figures 2 and 3 for a given value of q can persist even in a system with $n = \Omega(q^2)$ if b is small.

Though Fleet is largely functional at this point, its implementation is far from mature, and there are many opportunities for optimizations that we have not yet exhausted. We thus believe that future results will reveal better performance still.

9. Conclusion

The goal of Fleet is to provide a foundation on which critical and scalable distributed applications can be built. Fleet supports the abstraction of persistent shared objects, by which clients can communicate by performing method invocations on these objects. To support critical applications, Fleet provides protocols for building persistent objects that tolerate arbitrary (Byzantine) server failures, and provides the name space abstraction to limit interference between the objects of different applications. Fleet is fully extensible with new object types, which can employ the default Fleet method invocation protocols or other custom

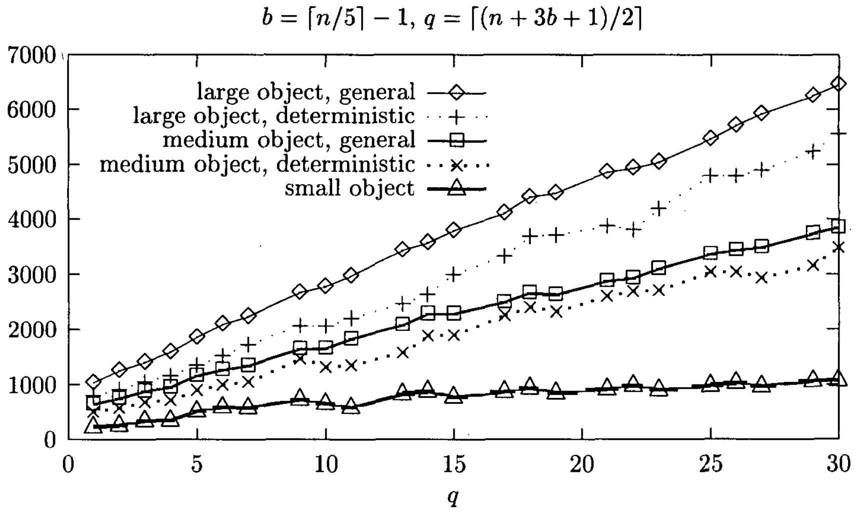

Figure 2. Single-client method response time (msecs) with quorums of size q

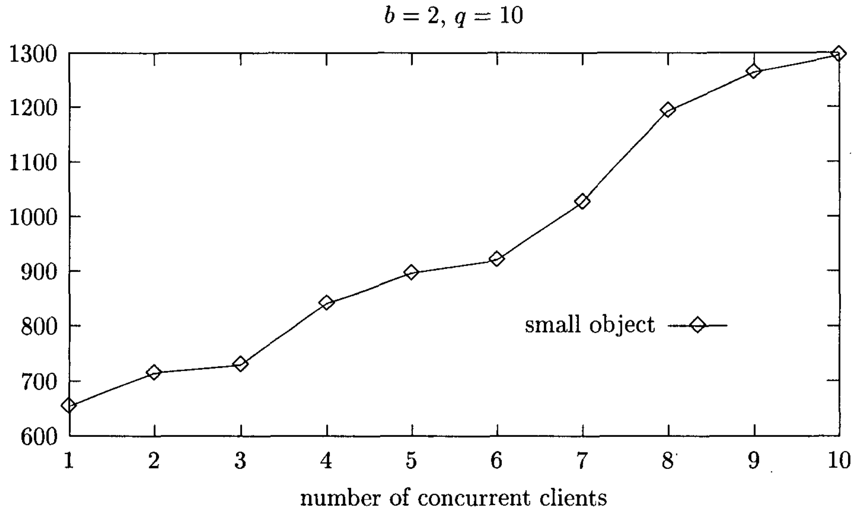

Figure 3. Multiple client method response time (msecs) with quorums of size $q = 10$

protocols. Those implemented using the default method invocation protocols enjoy strong (linearizable) concurrent semantics and a probabilistic approximation to the wait-freedom liveness property. Moreover, Fleet's use of Byzantine quorum systems for object replication enables an ability to scale that is unique among systems that can tolerate arbitrary server failures.

In this paper, we have described the current state of the Fleet system with an eye toward the application developer. We have described the basic components that comprise a Fleet object (FOHandles and FOReplicas), how Fleet objects are named (using FONames), and how new Fleet objects can be introduced into the system. We have also quantified Fleet's performance in our current implementation.

Fleet is an ongoing project that presents many opportunities for improvements and future research. Our ongoing work includes maturing the system for security, reliability and performance. We also continue to research ways to improve the semantics of Fleet objects, such as more effective ways of composing them.

References

[AMPR99] L. Alvisi, D. Malkhi, L. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. In *Proceedings of the 7th IFIP Working Conference on Dependable Computing for Critical Applications*, pages 357–371, January 1999.

[And96] R. Anderson. The Eternity service. In *Proceedings of PRAGOCRYPT '96*, Czech Technical University Publishing House, September 1996.

[BHL00] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[CEG+99] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries*, August 1999.

[CMR01] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, April 2001. To appear.

[CvE98] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, October 1998.

[HCC+98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[Her91] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13(1):124–149, January 1991.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463–492, July 1990.

[Kra93] D. W. Kravitz. Digital signature algorithm. U.S. Patent 5,231,668, 27 July 1993.

[Lam86] L. Lamport. "On interprocess communication (Part II: algorithms)". *Distributed Computing* 1:86-101, 1986.

[LCSA99] B. Liskov, M. Castro, L. Shrira and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, June 1999.

[LS99] J. Larsen and J. Spring. *Globe: Global Object Exchange*. Masters Thesis, University of Copenhagen, October 1999.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.

[MF99] G. McGraw and E. F. Felten. *Securing Java*. Second edition, John Wiley & Sons, 1999.

[MMR99] D. Malkhi, Y. Mansour, and M. K. Reiter. On diffusing updates in a Byzantine environment. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 134–143, October 1999.

[MMRT00] D. Malkhi, M. Merritt, M. K. Reiter, and G. Taubenfeld. Objects shared by Byzantine processes. In *Proceedings of the 14th International Symposium on Distributed Computing* (Lecture Notes in Computer Science 1914), pages 345-359, Springer, October 2000.

[MR98a] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4):203–213, 1998.

[MR98b] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[MR00] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

[MRW00] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing* 29(6):1889–1906, 2000.

[MRWW01] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic quorum systems. *Information and Computation*, 2001. To appear.

[RCW01] A. Rudys, J. Clements and D. S. Wallach. Termination in language-based systems. In *Proceedings of the 2001 ISOC Network and Distributed Systems Security Symposium*, February 2001.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, February 1978.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[TL98] P. Tullman and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, September 1998.

[WRC00] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident and censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso. "Understanding replication in databases and distributed systems". *In Proceedings of the 20th international conference on distributed computing systems (ICDCS 2000)*, pages 264-274, Taiwan, April 2000.