A Distributed Data Collection Framework

Design, Prototype, Implementation and Evaluation

M i c h a e l   S t o c k m a n

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-70

A Distributed Data Collection Framework

Design, Prototype, Implementation and Evaluation

Michael Stockman
Examiner (LECS/IMIT/KTH): Vladimir Vlassov
Co-Supervisor (SICS): Konstantin Popov

Master of Science Thesis in Programming Technology
Stockholm, Sweden 2004

IMIT/LECS-2004-70

Abstract

We have sketched on a distributed collection framework, similar to the local collection framework by Sun, using Java and peer-to-peer technology. A major goal of our work was to create collections that were distributed in the sense that fragments of a single collection could be stored on different nodes in the network.

A further goal that significantly influenced our work was having distributed objects in the collections using explicitly controlled caching and replication. We believe that we have provided an architecture, borrowing from Vrije Universiteit’s Globe project, where this can be explored in the future.

These goals are interesting to pursue today. Distributed collections containing data objects that represent physically distributed services could for instance have applications in Grid services. Explicitly controlled caching and replication of the system could increase performance of the system through tuning the system to the memory behaviour of the particular application.

We created a prototype implementation of the proposed design to evaluate parts of the design and provide a proof-of-concept. A part of this implementation is a list structure that can be partitioned over several nodes. We used Tapestry to provide peer-to-peer services. Our evaluation of this structure showed that the concept not only works, but can even be competitive with local list implementations under some circumstances and for very large lists. We see great potential for further optimization of our framework.

Through the evaluation we also gained some practical experience of working with the API in distributed settings. We have described some of this experience and suggested some alternative solutions in places where we found that the differences between local and distributed computing made the API awkward.

Abstrakt

Vi har skissat på ett distribuerat system för mängdhantering (distributed collection framework), liknande Suns system för lokala mängder, med hjälp av Java och peer-to-peer teknik. Ett viktigt mål var att skapa mängder som själva var distribuerade, i den mening att olika fragment av en mängd kan lagras på olika noder i ett nätverk.

Ett ytterligare mål som påverkade vårt projekt var att ha distribuerade objekt i mängderna som tillät explicit kontroll över cachning och replikering. Vi tror att vår design, som lånar från Vrije Universiteits Globe projekt, kan användas för att utforska det i framtiden.

Det här är intressanta mål att utforska idag. Distribuerade mängder som innehåller dataobjekt som representerar fysiskt distribuerade tjänster skulle till exempel kunna användas i Gridsystem. Explicit kontroll över cachning och replikering kan höja systemets prestanda genom att det optimeras för den särskilda applikationen.

Vi gjorde en prototypimplementation av den föreslagna designen för att kunna utvärdera delar av den och för att visa att konceptet fungerar. En del av den implementationen är en liststruktur som kan partitioneras över flera noder. Vi använde Tapestry för att tillhandahålla peer-to-peer tjänster. Vår utvärdering visade att konceptet inte bara fungerar, men att det också kan konkurrera med icke-distribuerade listor under en del förutsättningar och för mycket stora listor. Vi ser också stora möjligheter att optimera vårt system ytterligare.

Genom utvärderingen så fick vi också en del praktisk erfarenhet av att använda APIt i en distribuerad miljö. Vi har beskrivit en del sådana erfarenheter och föreslagit alternativa lösningar där skillnader mellan lokal och distribuerad databeräkning har gjort APIt obekvämt.

Acknowledgements

First I would like to thank my supervisors Vladimir Vlassov and Konstantin Popov, who initiated this work and have helped me through it. Thanks also to my opponent Lotta Rydström for many useful comments. Finally, I would like to take this opportunity to thank my family and friends for being there, not just during my work on this report, but all the way through KTH.

Table of contents

Table of Figures

Table of Tables

1 Introduction

In this report we discuss the development of an API for a distributed collection framework using peer-to-peer technology. We were also interested in experimenting with caching/replication and explicit control over the consistency mechanisms as a tool to increase performance. Finally we wanted to develop a prototype implementation to evaluate the API and architecture.

Grouping data objects together and maintaining those collections is one fundamental part of many, if not most, computer programs. The collections can then contain the cards on a hand in a game of poker, the hands themselves in a game of poker, the mails in a mail-folder, the UI controls visible on the screen, or virtually any other group of data objects. Before the advent of standard collection frameworks, such as Sun's Java Collections Framework [1] or (part of) standard template library [2], it was not uncommon for programmers to write their own collections, perhaps even several times, in each application.

Some collection frameworks thus became stunning successes and widely adopted. These restricted however the collections to only being accessible on a single computer and the collections could only use the resources of that single computer to store data. On the other end were the database systems, which often allowed access from many computers and sometimes even could combine the resources of several computers to store a collection. They did on the other hand offer different data access-paths from the main memory access-paths in popular programming languages like C++ and Java. They also did not offer fine-grained control over the consistency of data and collections.

Thus we wanted to explore a combination of these fields with knowledge from the field of distributed shared memory. From this we hoped to derive a collection framework with multiple access-points open to future experiments with relaxed coherence of the collection and data for high performance.

1.1 Method

We used the traditional master thesis method while working on this report, i.e. something resembling the waterfall method. First we researched previously performed work related to our own, most of which we describe in Section 2. After a while we begun sketching on our own design, presented primarily in Section 3, and thus cut back on finding more related work. When that started to come together, we started our implementation work. Details on that, lessons learned and our evaluation of the result is discussed in Sections 4 and 5.

We did however not put watertight seals between the stages in this work, which is sometimes associated with the waterfall method. We did to some extent move back and forth between stages, though the general direction was quite linear. We do not feel that this affected the results negatively.

1.2 To the reader

In Section 2 we introduce work related to our own. Our focus was on providing a feel for the most relevant results of those works and their thoughts behind them. The reader that is already familiar with other distributed shared memory systems, consistency models or protocols might want to skip these parts.

In Section 3 we detail our approach to the design of the collection framework API and its implementation. We think that the reader will find that reading Sections 4 and 5 will be easier if he has read Section 3 first. Sections 4 and 5, as well as later sections, can probably be read rather independently of each other.

1.3 Achievements

We have examined a number of local collection frameworks, distributed collection frameworks (in a wide sense), distributed shared memory approaches, consistency models and consistency protocols. Based on this we designed a collection framework with the same API as Sun's Java Collections Framework. Supporting this framework, we designed a distributed object framework, based on ideas from Globe [3], using peer-to-peer technology, in particular Tapestry [4]. This object framework can also be a first step towards future experiments with explicit control of caching and replication.

We have also made a prototype implementation of some parts of this design to evaluate. The most important results are related to the API of the collection framework where we found that some patterns used in the Java Collections Framework, and that work well locally, appear cumbersome both to implement and use in the distributed case. We also measured the performance of our framework, with regard to both speed and capacity of collections. To some extent we compared these numbers with measures from the Java Collections Framework, which is a different kind of collection framework while having a very similar API, and found that our framework performed well only for very large collections and indeed also could manage collections much larger than the Java Collections Framework.

2 Related work

In this section we cover work relating to our own. This includes software distributed shared memory in various forms, consistency models and protocols, different kinds of collection frameworks and some peer-to-peer technology. To some extent this section provides a view of our literature study and it definitively lays a foundation of knowledge that later sections build on. That said, the initiated reader might already be familiar with much of the work presented here.

2.1 Software Distributed Shared Memory

Software Distributed Shared Memory (DSM) is software that provides shared memory in a distributed system. Such systems always use message passing to coordinate the accesses to the memory between different nodes. The message passing protocols are called consistency protocols and implement consistency models that describe the meaning of read and write primitives, often in relation to some locking mechanism.

Distributed shared memory has in part been of interest to the computing industry because it offers an alternative abstraction for parallelizing and distributing algorithms. It is particularly useful in cases where message passing is cumbersome to use directly because it hides the details of the message passing and frees the programmer to work on other things.

The field became mainstream after Li and Hudak publicized their paper on the Integrated shared Virtual memory at Yale system [5] in 1989. Much research and development has been conducted on shared memory systems since then and now there are plenty of papers available on many variations on the theme.

One class of systems makes shared memory available to the application programmer like paged virtual memory shared with other programs. Another class of systems make the shared memory available as objects that the programmer can manipulate. It is also common to recognize a third class of systems, i.e. the tuple spaces, which are pretty close to being objects but not quite.

As pointed out by Waldo et al. [6], loosely coupled (distributed) systems are in some sense more parallel than even tightly coupled multiprocessors (parallel computers) from an application programmer’s point of view. They meant that threaded applications on a tightly coupled system have the option to coordinate at will and can take advantage of operating system services to communicate and recover from failures, while a distributed system without a single point of resource allocation, synchronisation or failure recovery is conceptually different. They described some of this difference as there being truly asynchronous operations in the distributed system while in the tightly coupled system the program doesn't get more parallel than the application programmer has opted for.

A choice that every software DSM has to make is what to do when a thread tries to access a part of the shared memory that is not directly accessible. There are three basic alternatives. The first is to send a request message and then halt until the memory arrives, i.e. fetching the data to the executing thread. The second option is to send the thread to the processor that has the memory and let it continue execute there, thus sending the executing thread to the data. The third option is to split the thread and let it execute at the data for a while but eventually return, usually some kind of remote procedure call.

2.2 Page-based Software Distributed Shared Memory

The first organization of software distributed shared memory that we will look at is the linear model. It can be integrated with the virtual memory system or it can more or less replace the virtual memory system (e.g. in a user-level implementation) and trap regular memory instructions, that giving them meaning on the shared memory. Another solution is to provide new instructions for manipulating the shared memory and copy to and from local (regular) memory, e.g. similar to file-handling.

This kind of memory is linear in the sense that an address represents a fixed number of bits while for instance object-memory often, but not always, address objects linearly but the objects can be of different size. To make the memory manageable it is usually broken down into pages, much the same way as with local virtual memory systems.

2.2.1 Integrated shared Virtual memory at Yale

Integrated shared Virtual memory at Yale (IVY) [5] was a system featuring paged shared virtual memory and which also could swap and share processes. The distributed memory was kept sequentially consistent (Lamport [7]) and coherence was maintained by a one-writer-many-readers protocol (in fact they evaluated several variants of it). They also discussed many implementation details that have remained central in distributed memories, such as the granularity of coherence, the coherence protocol, message sizes and overheads compared to the best sequential program.

After IVY that supported sequential consistency other page-based DSM systems with relaxed memory models emerged, such as Midway, Munin and TreadMarks. These relaxed memory models were developed at roughly the same time, notable examples include processor consistency (Goodman [8]), weak consistency (Dubois et al. [9]), release consistency (Gharachorloo et al. [10]) and entry consistency (Bershad et al. [11]).

2.2.2 Midway

Midway supported several consistency models and let the programmer chose between processor, release and entry consistency for the data in the program using special annotations, which Bershad et al. described [11]. The idea behind supporting several coherence protocols was to let the programmer quickly develop programs in a stronger model and then gradually insert annotations to let the system use weaker coherence protocols for more and more of the data and thus gain in performance.

2.2.3 Munin

Munin, by Carter et al. [12], supported only release consistent page based shared memory. Instead Munin allows the programmer to select between different implementations of the coherence protocol favouring different data flow patterns in an application, such as producer-consumer and migratory data. This did allow Munin to move pages in advance to where they most likely would be needed and invalidate pages that were being moved and most likely would not be needed again, and could thus potentially both reduce the number of page faults and messages.

2.2.4 TreadMarks

TreadMarks, by Keleher et al. [13], also only supported release consistency. They borrowed the multiple writer protocol concept from Munin but used it in a lazy release consistency protocol, while Munin’s protocols were eager. The purpose of this was to reduce communication.

The implementation of ThreadMarks was entirely user-level. They found that Unix communication was the limiting factor for them. To minimize that they used unreliable network protocols (UDP, AAL3/4) and, as needed, implemented specific protocols on top of them to ensure delivery.

2.2.5 Shasta

Shasta by Scales et al. [14], as well as Khazana below, were different from the previous shared memory systems. Instead of being integrated with the virtual memory system these systems used other means to maintain the consistency of the shared memory. In Shasta this meant that a program modified the executable code and inserted necessary checks and calls around accesses to shared memory.

Shasta provided a release consistent memory with potentially much smaller coherence objects than the other shared memory systems mentioned here. They divided the shared address space into lines of typically 64 or 128 bytes. Coherence was maintained on block level, where each block was made up of one or more lines[1]. A particular feature of their system was that not all blocks had to be of the same size, although the line size was fixed at compile time. This allowed the programmer to allocate memory with fine-grained coherence for structures sensitive to false sharing while at the same time using coarse-grained coherence for other structures.

2.2.6 Khazana

Khazana by Carter et al. [15] also was not integrated with the virtual memory system. Unlike Shasta it instead provided the abstraction of a separate shared memory space and provided primitives to the application programmer for explicitly accessing it.

The system is open with respect to consistency model. Pages of memory are associated with a region that can be locked. The region is then associated with a consistency manager which is responsible for updating pages and coordinating with other consistency managers as required before granting the lock, and when it is released. Their default coherence model is single-writer entry consistent with a predefined binding between lock and coherence object.

Susarla et al. [16] described their experiences with an extended system which supported C++ objects in the shared memory, as well as had support for event notification and allowed some user control over update propagation. These were all features they found very useful when they ported programs using pointer rich data structures and interactive programs.

[1] The software checking approach was a prerequisite for this fine grained approach to be viable. The virtual memory hardware could only efficiently be used to maintain coherence on page-sized objects, and pages were much larger than that.

2.2.7 JIAJIA

JIAJIA was a distributed shared memory by Eskicioglu et al. [17]. It used the scope consistency model previously introduced by Iftode et al. [18], who relied on simulations when they presented it and compared it in particular with the release consistency model. JIAJIA did not use the kind of special hardware to send memory diffs that had been assumed in those simulations. They did however achieve results comparable to TreadMarks, which now had had a few years to mature, in their tests. In other tests by Hu et al. [19] JIAJIA performed well against CVM after having more optimizations added. Still, its coherence protocol was pretty straightforward.

2.3 Distributed Object Memory

Distributed Object Memory (DOM) was developed in parallel with paged memory. Page-based DSMs received the most attention for some years but the focus has gradually shifted in favour of DOM. This kind of system often has small coherence objects compared to a page-based DSM system. The data within a single coherence object has also been grouped manually by a programmer and is thus often semantically related. In contrast on page-based DSMs the data on a page was often automatically laid out by the compiler and could thus be unrelated. Keeping related data together and unrelated data apart is thought to help reduce false sharing. Objects encapsulating data, that can only be accessed through method calls, also provide the software memory system with convenient points to insert the communication and bookkeeping code.

2.3.1 Emerald

One of the early distributed object memory systems was Emerald by Jul et al. [20]. It was a programming language and supporting run-time system. It featured an object model somewhat different from the big languages of today. The most significant difference might have been active objects, something which contained code that started to execute on a new thread upon creation of the object.

Otherwise, it featured both mobility of data and execution between the nodes in a cluster. They did still put a lot of emphasis on that execution not requiring threads and data to be moved should comparable in speed to systems without distributed objects (such as plain C). Removal of unused objects was achieved by garbage collection in two variants, one for node local objects unknown to other nodes (expected to account for a majority of objects) and one coordinated across the cluster. Both collection algorithms were based on mark-and-sweep although modified allow the system to perform normal work concurrently.

2.3.2 Amber

The Amber system by Chase et al. [21] was derived from PRESTO, a system for parallel computers by Bershad et al. [22]. Amber was designed to execute a single program, produce a result and then terminate. It therefore didn't include support for persistent objects, reliable computing nor communication between unrelated processes. It did provide mobility of execution and data, explicit locks, a globally shared object space and co-location of objects. It only allowed objects marked read-only to be replicated on several nodes. It did not automatically move data between nodes, but provided the application with explicit primitives for that.

Amber was built in and for an early version of C++. There was no difference between shared and local objects, all objects were shared. It was even so that all object pointers were valid on all nodes, so they could easily and safely be moved between. Objects were written as regular C++ classes by the programmer and the system provided a special precompiler to add the bookkeeping. As the system never transmitted objects automatically and only performed local method invocation, it would move the thread to the object. Because of this they also didn't serialize method calls but left it to the programmer to use locks and the ordinary memory system to manage coherence between concurrent method invocations.

2.3.3 Orca

Orca was a language for distributed object memory by Bal et al. [23]. It aimed at being suitable for general application programming through clean and simple semantics. It thus supported strong encapsulation only allowing class data to be accessed through user defined methods. On top of this they had a strong synchronization model requiring that all method invocations appear to execute indivisibly. They only allowed method invocations to block initially on designated guard conditions, thus possibly forcing the run-time system to roll-back a partial method invocation if it would block anywhere.

Another feature of the system was that graphs of objects were explicit first class objects rather than implied by reachability. They found that this simplified the situation when graphs were replicated between nodes, it made assignment of graphs a well-defined operation, when graphs were passed as in or out parameters and during deallocation.

2.3.4 C Region Library

The C Region Library (CRL) by Johnson at al. [24] was not written and/or used with an object-oriented language. It has been placed in this section because it had an object address space in which each shared memory area of arbitrary size occupied one address. Coherence was maintained on area level so they were for our purposes very similar to objects, even without other characteristic object oriented features like encapsulation or inheritance.

Coherence was, as already mentioned, maintained on area level and claimed to be similar to entry or release consistency. Unlike other such systems at that time the programmer did not provide the synchronization objects but they were provided by the system, one reader-writer lock for each area. In practice the programmer was responsible for locking every memory area before accessing it, just like he had to explicitly control which objects would be mapped in the program's local address space at any time.

2.3.5 Common Object Request Brokerage Architecture

Common Object Request Brokerage Architecture (CORBA) is a standard for distributed object frameworks that emerged out of the industry via the Object Management Group. It aims very much at isolating the user of an object from its implementation by only allowing access through well-defined interfaces [25]. Thus it has defined a distributed object model independent of implementation language. On top of this model they have defined the interface of many higher-level CORBA services. Some of these has been criticized for being to vague to actually provide any implementation independence, such as that by Kleindienst et al. [26] of the persistent object service in relation to the relation service and the externalization service.

2.3.6 Distributed Computing Environment

Distributed Computing Environment (DCE) from The Open Group [27] was originally a framework only for remote procedure calls in a client-server manner. Once object oriented programming had made its break through in the industry, marked by for instance the success of object-oriented languages and the adoption of CORBA, they felt that they had most of the features that characterized an object framework and should now add the rest.

This was for example completed in the DC++ framework [28] where full object support was added with C++ while remaining true to the DCE framework and its standardized system services. This was their second attempt at creating a distributed object framework, the first of which had not used DCE but raw network services. They found, amongst other things, that the high-level, standardized system services provided by DCE provided a crucial advantage for making a stable environment quickly.

2.3.7 Java Remote Method Invocation

Sun's Java Remote Method Invocation (RMI) [29] has been the distributed computing framework shipped with most if not all Java programming environments since very early in the Java evolution. They described it as a remote procedure call mechanism at its most basic level. At the same time it takes advantage of many features of the Java architecture to provide mobility of code and data, in this case code can even be dynamically added to a program image at run-time.

RMI distinguished between three kinds of Java classes, remote classes, serializable classes and other classes. The first kind are the server objects that essentially service remote procedure calls. Instances of both the first and second kind can be arguments and return values in these procedure calls. Instances of the second kind are then copied into the server object’s address space but no relation is maintained with the source objects on the client side. Instances of the first kind do instead result in a stub object being created on the other side that forwards accesses as procedure calls. The third kind of instances cannot be transmitted as part of a remote procedure call, which would not be meaningful for instances containing transient process specific data like file or window handles etc.

2.3.8 Globe

Globe, by van Steen et al. [3], was an object framework particularly targeting the Internet and distribution of web objects. Still they aimed at making a general object framework, and thus had to design both for objects that must be coherent as well as objects that could be allowed to be somewhat incoherent.

This resulted in that they divided the local part of a shared object into four parts (sub-objects). Two of these were specific to the object’s *class*, the control sub-object and the semantics sub-object, and two were more general, the replication sub-object and the communication sub-object. The control sub-object was the facade to the application. It used the replication sub-object to synchronize with other local objects of the same shared object, of course using the communication sub-object. The class specific object data and code were in the semantic object. This division of concerns obviously allowed separation of the class’ business logic from distribution issues, both issues that had to be dealt with but that now could be dealt with separately.

They used an implementation in C for their prototype but for their real version they chose Java and RMI [30]. For object naming they used DNS records, providing a global object pointer, which then was resolved into object locations. They found amongst other things that using Java had not been a problem for performance and that it must be easy both to write applications and to use the supporting software.

2.3.9 Jini

Jini [31] is the latest in the Sun family of Java distributed object frameworks. It builds on previous work, in particular the Java environment and RMI, and extends them with a notion of services and a basic set thereof. Like CORBA it aims very much at connecting service users with service providers, the key features being joining the network, finding resources in it and lock, transaction and event processing. Like in CORBA they have chosen to leave significant latitude to those creating implementations and/or higher-level specifications on top of jini. For instance the meaning of transaction and the semantics of a transaction are left unspecified although the ACID model is recommended, which in turn has different conformance levels.

2.4 Tuple Spaces

Tuple spaces are a third approach to shared memory next to shared memory and shared objects. Its conceptual model is that a process wishing to share information with other processes packs it into a tuple and make it available in a common space. Another process interested in that information can then take the tuple from the space. Alternatively tuple spaces can be viewed as a generalization of message passing with the space as the global communication area, tuples as messages and processes free of the requirement in message passing to name the recipient/sender of every message [32].

The original primitives in tuple space systems were `eval` (to create a process), `in` (to read and remove a tuple from the space), `out` (to write a new tuple to the space) and `rd` (to peek at a tuple in the space). `eval` and `out` were asynchronous operations while `in` and `rd` were synchronous. `in` and `rd` supported pattern matching on the tuple in order to only operate on a matching tuple, as well as blocking until there was a matching tuple available. Some later tuple space systems has expanded somewhat on the primitives available, perhaps primarily in attempts to raise performance in different cases.

In our context it is interesting to note that these primitives have nice properties for our application, tuples in the tuple space are read-only and when an object (tuple) is taken from the space by a process other processes automatically block until it is put back. Read-only things are nice in a loosely-coupled environment, like ours, since it is easy to maintain copies of them consistent. Still remains the problem for the tuple system to be consistent about exactly which tuples are in the space though. Blocking for a tuple can be considered to have essentially the same effect as a lock on an object would have had.

2.4.1 Linda

Linda [32] was a distributed computing model by Gelernter and pioneering the idea of tuple spaces. It was not intended as a system on its own, but as a "plugin" into other computing models dealing only with process creation and coordination. Thus a Linda system could belong to any computing approach, such as object oriented languages, logic languages, be an interpreted language, and so on depending on the particular system to which the Linda primitives were added.

The generalized messaging view on tuple space programming mentioned above was one of the central ideas in Linda. It decouples producers of data from consumers of it compared to traditional message passing. The programmer writing a data producing process is thus relieved of much thinking about consumer processes and can to a large extent just drop the data into the space and trust that things will work out. In the paper this simultaneous thinking about producers and consumers was referred to as "thinking in simultaneities".

The paper studied not only described Linda systems, but also did so in context of alternative approaches to parallel programming. Doing so they spent considerable effort questioning the "truth" that parallel programming is inherently difficult and challenged that this depended on problems with message passing, concurrent objects and concurrent logic programming. They also saw concurrency problems with the functional languages of the time, as they entirely relied on the compiler choosing an appropriate level of granularity and that capability had not been demonstrated in practice. Still their experience was that parallel programming need not be very difficult if using the Linda (their own) primitives, explicitly parallel and with simple semantics. With recent initiatives like JavaSpaces maybe they are finally beginning to reach larger audiences in the industry.

2.4.2 JavaSpaces

JavaSpaces is a specification for a Linda like tuple space service on top of jini [31]. The tuple space consists of records that can be of either primitive or class type. They can thus be used to share both data and code, since a value of class type can contain methods. A JavaSpaces compliant implementation is also required to use the jini transaction mechanism to provide ACID style transactions, and the distributed event mechanism to allow clients to monitor changes to a Space. Sun distributes an implementation of JavaSpaces called outtrigger with its jini starter’s kit. GigaSpaces below is another example of a JavaSpaces implementation.

IBM has a project called TSpaces [34] which is very similar to JavaSpaces. It is however not built around jini and has a somewhat different API, so they are not directly interchangeable for each other.

2.4.3 GigaSpaces

GigaSpaces by GigaSpaces Technologies Ltd [35][36] is as previously mentioned an implementation of JavaSpaces. In addition to the basic tuple space operations it aims to also provide administrative features necessary in corporate environments, such as clustered spaces with advanced replication, caching and load-balancing features and various security features.

The foundation of GigaSpaces are the local spaces maintained by each server. These can be independent but can also be connected to spaces on other servers, either as peer-spaces that together store the tuples or as master-cache spaces. A particular kind of cache space is the kind connected to a cluster of master spaces which has the ability to switch to another master space if the current would become unavailable, and also to participate in load-balancing. It is also possible to prioritize the master spaces such that the cache space will prefer near spaces to remote spaces. Further it is possible to in detail control how updates propagate through a cluster. Caches can also use invalidate- or update-based protocols as configured for the particular application.

GigaSpaces is thus a highly versatile product. On the other hand this also means that the application developer/user has to make a lot of complex configuration decisions that, just as they could boost performance, also could kill it. This is also reflected in their general recommendation to not only buy their system but also hire them to configure it.

2.5 Object databases

As the object-oriented languages grew in popularity also the database research and vendors followed. This meant that both the data storage facilities and query languages had to evolve to provide features expected from an object-oriented environment. The field eventually fell into some disgrace as the industry, for pragmatic reasons, tired of purists within the research community and moved towards the object-relational systems that are common today.

Structured Query Language (SQL) was the prevailing relational query language at that time. Since the release of the second SQL revision 1992, the evolution of SQL has progressed towards support for persistent complex objects including abstract data types, object identifiers, inheritance, polymorphism, encapsulation, etc. SQL is however far from the only language for manipulating and querying object databases. Some databases provide APIs for different programming languages, some are fully integrated with the execution environment (either through being linked with the application or through loading the application and running it), yet others define their own query and manipulation languages.

2.5.1 Oracle

The Oracle Database System [37] was originally a relational database system, but has been extended with many features including objects and object tables [38]. Their object model is quite complete with both data and methods, although encapsulation isn't supported. It belongs also with the transaction databases, so transactions on objects and tables are constrained by the ACID[2] model, where they by default allows reading committed data from other concurrent transactions in a transaction.

They also have mechanisms to partition and/or replicate data between several database servers [39]. They essentially have two levels of replication that can be used in a mixed environment, either cooperating peers (master sites) or master-slave (materialized views). Both of these requires some knowledge and work from the user, both to get going and sometimes also to recover after a failure.

2.6 Consistency protocols

The purpose of a consistency protocol is to implement some consistency model, which essentially defines which value a read of any part of the shared memory may return. Another way to look at it is to reason about the allowed orderings of events (accesses to shared memory) that may be observed by the processes in a multiprocessor.

After choosing a consistency model the protocol designer is also free to make a number of other decisions that does not directly affect the programmer's view of the memory, but that might affect the performance and/or the complexity of the implementation of it. This can be such things as if coherence objects have a home node, if that might change during execution, or if they don't.

[2] Atomicity, Consistency, Isolation and Durability.

2.6.1 Sequential Consistency

On a uniprocessor it is intuitively assumed that a read will return the last value written. In a distributed environment a more strict definition is needed, since it might not be intuitively clear which value is the last one if two processors write to the same part of the shared memory. Thus in 1979 Lamport published the now classical definition of sequential consistency

> *"A multiprocessor is said to be sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*
>
> *- Leslie Lamport [7]*

This were to become the standard against which other consistency models were compared. On such a system each memory operation appears globally atomic, which makes the intuitive notion of last value written work again. This simplicity did however come at a price

* Systems implementing sequential consistency were sensitive to false sharing, where a part of the memory is sent back and forth between some processors writing to it simultaneously (known as the ping-pong effect).

* Compiler optimizations involving instruction reordering could not be used since the correctness of the program could depend on the particular order certain instructions were issued.

It was however known that many programs did not rely on the memory being sequentially consistent but instead used explicit synchronization mechanisms such as locks and barriers to protect several processes from accessing the same memory simultaneously, as was observed by Goodman [8] and by Bershad et al. [11]. Goodman also pointed out that this is also generally considered good programming practice. The result that they wanted to take advantage of is that these programs will execute on a multiprocessor with a weaker consistency model as if it had been sequentially consistent.

2.6.2 Processor consistency

Goodman built on that when he suggested the processor consistency model [8] and a hardware architecture. It relaxed the requirement that every instruction must be totally ordered (with respect to every other instruction - sequential consistency) to that it only must be partially ordered (ordered with respect to other instructions executed on the same processor). The architecture would allow greater memory-access parallelism in processor consistency mode than in sequential consistency mode, thus reducing the effects of memory latencies. Virtually all programs would still run correctly on it, except for those using what he called pathological cases.

2.6.3 Weak consistency

An even weaker consistency model was already known then, since Dubois et al. had already published their paper introducing weak consistency [9]. The emphasis was towards hardware shared memories also in their paper. For implementations of this, as well as in the yet weaker consistency models, it (so far) has been necessary for the implementation to be aware of the synchronization in the program. Executing a synchronization instruction (such as acquiring or releasing a lock) often signifies a change in which shared memory the process is supposed to access, and whenever that changes it is always precisely at a synchronization instruction[3]. The general idea is to take advantage of this by allowing memory that the process is not supposed to access to be inconsistent. This does not necessarily require the consistency protocol to know exactly which memory the process is supposed to access to work.

2.6.4 Release consistency

The consistency model that in the end won the hearts of software shared memory designers was release consistency, which was first presented by Gharachorloo et al. [10]. It is an even weaker model than weak consistency. In the weak consistency model memory accesses are considered either ordinary or special (e.g. when used for synchronization). Release consistency further refines this by dividing special instructions into synch (when used for synchronization) and nsynch (other), and even divides synch instructions into acquires and releases.

Acquiring a lock would for instance be an acquire instruction, and it can at that time reasonably be expected that the process now may access shared memory that it shouldn't before, and thus any local caches must be updated with remote updates. If several processes has a lock to the same data then they are expected to only read. Similarly when a lock is released then that would be a release instruction, and then there may be some shared memory that this process is no longer supposed to access but that another process may access after performing an acquire. Thus all processes performing an acquire after the release must see the changes (if any) made by this process to the shared memory. After is governed by that special instructions are required to be processor consistent.

Finally they showed that any properly labelled program would execute as though it was executed on a sequentially consistent multiprocessor on a release consistent multiprocessor. Properly labelled essentially means that there is a sufficient number of release and acquire instructions for this to be the case. A special case of proper labelling is a synchronized program where all writes to shared data is done in critical sections (no other reads or writes can occur simultaneously), then all accesses except the synchronization primitives can be considered ordinary.

2.6.5 Entry consistency

When Bershad et al. then created entry consistency [11] they built on release consistency. As already told release consistency requires all shared memory to be consistent after an acquire and writes to all shared memory to be made visible to other processes after a release. By adding information about exactly which shared memory each synchronization object governed access to, they could further relax the model to only require that that particular memory be made consistent when an acquire or release is performed.

While many parallel programs were synchronized, which was good programming practice, they did not contain this additional binding between memory and synchronization objects. Thus porting a program to this model might require a bit more work than would have been the case with any of the previous models.

[3] Not just because these models require it to be at a synchronization instruction, but also because that is good programming practice as previously mentioned.

2.6.6 Scope consistency

To ease porting while still reaping the benefits of a weaker memory model Iftode et al. introduced scope consistency [18]. They introduced a concept called consistency scopes, something that can be likened to all critical sections being protected by the same lock. There is also a "root" consistency scope covering the entire program. When a process enters a consistency scope it starts a new session, which then ends when the process leaves the scope.

The idea is that modifications to memory need only be visible within the scope that they occurred. Consistency is then maintained by the system associating the modified memory with all open sessions. Thus when the process closes any of the sessions (for instance by releasing a lock) then those changes must be made available to other processes, although they don't need to fetch them until they open a new session for any scope the memory is associated with. When that happens (for instance when acquiring the lock protecting the memory) the other process will see the previous changes.

Obviously the memory will become associated with other locks a process is holding while writing the memory (if any) that might be meant to synchronize other things. This might cause some more consistency work than with carefully managed associations in entry consistency. It might still be less than with release consistency, which is like every scope protecting all shared memory, and the programmer doesn’t have to explicitly manage the lock-memory associations.

The result is that scope consistency is a weaker model than release consistency. While they concluded that many programs assuming release consistent memory also would work with scope consistent memory, i.e. execute as if the memory had been sequentially consistent, it is not true for all. The natural migration path to scope consistency is using lock based scopes, and these differences might require critical sections to be extended causing more contention on them. They also provided for explicit scope annotation as an alternative.

2.6.7 Home based protocols

One choice in the coherence protocol is as previously mentioned whether if coherence objects should have a home node. In the IVY system [5] they used a home based protocol and split the pages among the participating nodes to get a fixed home for each page. They had concluded that using dedicated nodes as homes for all pages would essentially lead to it not being used fully until there were sufficient non-home nodes and then it would become a point of contention, i.e. a difficult balance problem, and not explored it further. Still, having fixed home nodes means that the home of a page might be and remain very far from the nodes actually using the page.

Another more extreme solution used for instance by TreadMarks is that coherence objects doesn't have home nodes. Obviously the home node cannot become a point of contention in such a scheme, but currently it seems they are more complex and doesn't scale as well when the number of nodes grow, at least if the protocol choses home nodes for the pages well [33].

A choice somewhere in between are migrating home protocols or directory based protocols. In these a page does have a home or manager, but since it can change other nodes might not know exactly where it is all the time and thus they have some mechanism to find it. This can for instance be a forward pointer to a node with better information, or a directory entry.

2.6.8 Eager protocols

A path that has been explored to reduce the communication needs of the consistency protocols is to send messages as late as possible (lazily) in which case it might turn out they will not be needed at all. TreadMarks used such a variant in its release consistent protocol, and it has been argued that it is most often, though not always, better than eager release consistency [40].

2.6.9 Simultaneous writers

False sharing is a problem that can haunt shared memory systems. It occurs when several processes access different parts of a coherence object and neither process wants to access a part that another process is writing. Then it would not be necessary to keep the memory consistent between the processes, but since the accesses are to the same coherence object the memory system cannot just assume that it is not true sharing.

Thus in systems with a sufficiently weak consistency model, like TreadMarks' or Munin’s, the coherence protocol allows each process to either read or write essentially any memory but does not use the accesses as points where the memory must be consistent. Instead they collect the individual changes made to each coherence object and then when the coherence object must appear consistent all changes are merged together. There are also versions that instead of collecting the changes propagate them asynchronously to other holders of the object or to its home node. If the application programmer got it right then there will be no collisions that cannot be ordered, and otherwise then there is a data race in the program and the result will be unpredictable.

Thus allowing simultaneous writers requires that memory is not implicitly synchronized on access and that changes within a coherence object can found and applied at other nodes while not touching other parts of the coherence object that didn't change.

2.7 Collection Frameworks

Collection frameworks have become a standard part of essentially every modern object oriented environment today. They are so appealing since they offer a reusable, trusted solution to the frequent problem of keeping track of collections of objects, instead of every programmer developing new collection management routines for in worst case every collection in every application.

Collection frameworks thus has the potential to reduce the development effort by providing a solution to a common problem and eliminate the bugs that would have been created while implementing an in place solution. They can also provide performance benefits when an efficient implementation of "the right" data structure is used, rather than even an excellent implementation of the data structure the programmer is most comfortable with. These advantages has also likely contributed to the success of for instance the Java Collections Framework.

A collection framework usually consists of a number of collections. A collection is an object on several different levels. On the implementation level it is usually an instance of a language dependent class construct, possibly also delegating parts of its responsibilities on to other instances. On this level they can also be considered realizations of one or more data structures and often provides access to implementations of common algorithms on those data structures, such as insertion, deletion, traversal, etc. On this level it is also valuable to consider the mathematical properties of a collection, such as if it is a bag or a set, if its elements have some structure and if its elements are ordered somehow. On a semantic level a collection represents the grouping of the elements in the collection. That can for instance can be interpreted as a deck of cards or a folder of mails.

2.7.1 Java Collections Framework

The Java Collections Framework (JCF) [1][41] is a standard Java component and a part of every Java distribution. It provides, among other things, a number of data structures wrapped in classes together with operations on them and is greatly appreciated by the Java community.

The design of the component is inheritance based and the inheritance graph primarily consists of two tree-like structures, which can be seen in Figure 33 and Figure 34. These can be seen as a super-inheritance tree of pure virtual classes (interfaces) and some sub-inheritance trees of implementations classes. The implementation classes also inherits from (implements) the interfaces. The difference between the two structures is that one contains classes implementing set or bag data structures and the other classes implementing binary relation data structures.

This approach has made it possible to write algorithms on data structures without being aware of their implementation, through programming by the interfaces. Function objects are not used extensively in this framework, but can be used to provide an ordering relation to some data structures.

Looking first at the set and bag hierarchy of JCF (Figure 33) we noted that all concrete implementations of `Collection` are also implementations of either `List` or `Set`. All elements in a collection can be examined using routines common to all collections, but on this level elements are not necessarily guaranteed to be kept in any particular order. There are also optional operations for adding and removing elements but only by value and not by any kind of position, and collections are free to throw unchecked exceptions e.g. if they for some reason don't like the value or if they are read-only.

`List` lists refine the guarantees of collection to maintain the elements ordered in a linear sequence. Elements can thus be indexed by their position in this sequence, and it makes sense to add operations for manipulating the collection using indices here. Still the operations that in some way modify the Collection are optional and may throw unchecked exceptions. Lists are also required to support a more powerful bidirectional iterator (`ListIterator`) compared to other collections.

`Set` sets on the other hand does not add new operations or stipulations about the contained elements other than that a value can obviously only occur once. A special kind of sets are the `SortedSet` sets that refines sets by promising to maintain the elements in either their "natural order" or the order defined by a function object supplied when the set was created. It thus made sense to add methods to query the function object, as well as to get subsets containing ranges of elements using the ordering.

Concrete `List` collections are `ArrayList`, `LinkedList`, `Stack` and `Vector`. `Vector` is a growable array that remains from the time before JCF, but was retrofitted to become a part of it. Herein lies the reason it contains many redundant operations. `Stack` is an extension of `Vector` and adds the classical stack operations to it. `ArrayList` is the new growable array implementation that is consistent with the JCF design. `LinkedList` implements `List` on a doubly linked list and is thus suitable to be used e.g. as a stack or a queue.

There are three concrete `Set` classes: `HashSet`, `LinkedHashSet` and `TreeSet` (which is the only `SortedSet`). These essentially just implement the operations specified by their interfaces (except for constructors that cannot be specified by interfaces) and does not add more operations like all of the `List` implementations. `LinkedHashSet` also orders the elements in the set, like `TreeSet`, but using insertion order instead of sorting the elements.

Then we looked at the binary relation (`Map`) hierarchy of JCF (Figure 34). It is useful to remember that a map can be seen as a set of tuples each containing one element from the key and value domain and let the meaning of a tuple in the set be that the elements are related. As then might be expected the `Map` hierarchy is similar to the `Set` sub-hierarchy, with a `SortedMap` similar to `SortedSet`.

Similarly to in the other hierarchy, the read-only operations are mandatory while the modifying operations are optional and might throw unchecked exceptions. The actual operations supported by maps are also very similar to those supported by sets, considering that maps work with key-value pairs. This similarity shows also in that sorted maps can create sub-maps much the same way as sorted sets can create subsets.

There are two main concrete classes in this hierarchy: `HashMap`, a regular `Map`, and `TreeMap`, a `SortedMap`. `IdentityHashMap` and `WeakHashMap` are both special purpose `Map` classes with odd properties. This hierarchy also has some legacy from the time before JCF. `Dictionary` is an old interface that has been superseded by `Map`, and `Hashtable` relates to `HashMap` very much like `Vector` relates to `ArrayList`.

2.7.2 Standard Template Library

The Standard Template Library (STL) [2] is a C++ framework available in most C++ development kits. STL is not inheritance based like the Java Collections Framework or to some extent the .NET collections framework, but instead based on the powerful generics capabilities of C++.

The foundation of the framework is the bag classes (Figure 36) and the relation classes (Figure 35). These use iterators (Figure 38) to a much greater extent than for instance the Java Collections Framework. This can be to return results of queries or as function arguments to denote a position in a collection or to act as a data-source.

The framework also has compositional adaptor classes (Figure 37), i.e. classes that build on some collection to provide a new interface. Using the C++ generics mechanism these can build on any collection exposing the required API. Thus it is not less powerful than Java where the same thing would be accomplished by the class being enclosed implementing an explicit interface that the encloser would be written against, but perhaps less self documenting.

The framework also provides a set of independent collection operations, not tied to any specific kind of collection. These are heavily based on using iterators as data sources and the passing of function objects to control the details of the algorithm, e.g. to define transformations on objects, to determine which object is sought or to define an order between objects.

2.7.3 .NET Collections Framework

The .NET Collections Framework [42] is similar to the Java Collections Framework with a interface hierarchy and subordinated implementation classes shown in Figure 39. It provides fewer data structures than its Java counterpart, but the key difference may be in the organization of the class hierarchy. In this framework the interfaces for both bag/set structures and relation structures are contained within a single hierarchy. At the same time the interfaces are significantly weaker in terms of the operations that can be accessed through them and it is more common for implementation classes to offer more operations than specified by its interfaces.

Mirza, involved in the .NET project, suggested a different approach in [43]. In this approach he identified various properties of collections. He then created collection types that supported different subsets of those properties (Figure 40). Finally the intention is to use composition to build new collections with new interfaces out of existing collections. Using composition rather than inheritance to do this obviously provides greater freedom to chose the interfaces the collection will support but as Mirza also points out this can also slow down the framework considerably.

Mirza also criticized the hard binding between programs and implementation classes that becomes the result of approaches like JCF or STL. He instead advocated complete use of factories to completely decouple them. Although these points appear valid, it is not clear that they are a problem to the extent that he suggests.

2.7.4 Active Collections Framework

The Active Collections Framework (ACF) and experiences from several implementations of it was described by Raj in [44]. It builds on the idea that data is stored in underlying data stores and then active collections are defined on top of those by defining inclusion predicates. The collections are active in the sense that they change as data changes in the underlying collections change and they make the application aware of the changes through an event mechanism, rather than forcing the application to poll for changes. Raj remarks that these inclusion predicates essentially become continual queries when they are applied to the data stores at run-time.

Raj also describes some practical experiences from working with ACF, primarily a CORBA implementation although several previous partial implementations had been made in various environments. That implementation was structured as a middle tier between the client tier and the data store tier. He notices amongst other things that the CORBA event service was inadequate for ACF as well as that continuous reevaluation of the active collection predicates was too inefficient. He concluded in the end that the ACF often has been a good foundation for building distributed applications.

2.7.5 Java Database Connectivity

Java Database Connectivity (JDBC) [45][46][47] is Sun's Java framework for accessing tabular and relational data. It is intended to let application programmers easily use SQL to communicate with data sources. Thus, it provides components for handling and manipulating SQL statements as well as a mapping between SQL types and Java types.

Apparently JDBC is quite integrated with SQL, as all JDBC compliant drivers must support all SQL-92 entry level commands. However a driver is free to also let the application programmer use SQL extensions or even commands from entirely different languages.

Looking at the collection framework properties of JDBC is not entirely easy as they depend on the capabilities of an underlying datasource. In most cases it should not be too far from the truth to look at it like a collection of tables that can contain rows. Rows can usually be added, removed, changed and fetched, but there are normal conditions when some or all of these operations may be restricted.

The operations are realized by the program providing the command as a string to the JDBC framework and executing it in the driver or an underlying datasource. There is also a component for accessing a result from e.g. a query command with the appearance of a table consisting of rows and columns.

2.7.6 Java Data Objects

Java Data Objects (JDO) [48] is a higher level Java service than JDBC, also aiming at facilitating data storage. JDO frees the application programmer from some of the details of data storage, such as using SQL which he would have to do using JDBC, but at the expense of some control over the process.

2.8 Distributed Hash Tables

Distributed Hash Tables (DHTs) is a an area of peer-to-peer distributed computing that has evolved rapidly during the last years. One reason for this interest is that fast computer networks and the ever more powerful computers connected to them have become usual and there is a desire to take advantage of spare resources. Another reason is the discovery of space and time efficient algorithms for peer-to-peer implementations of DHTs. Because of the size of some of these networks failure of components may be very frequent, which also has made fault tolerant peer-to-peer solutions very popular.

Some DHTs work on two levels. On the bottom level, the overlay network level, they can then provide message routing between the participating nodes. The actual hash table can then be built on top of that layer using it to route lookup and other requests to the right node. Once there it can easily service them and send a reply.

Other approaches to peer-to-peer computing include broadcasting lookup requests to all nodes on the network (like Gnutella [49]), using reinforcement learning strategies to specialize part of the network on parts of the key space and finding these parts (like FreeNet [50]), various forms of hybrid solutions containing either entirely centralized resources (like Napster [51]) or unequal peers (like Gnutella2 [52]) and virtual distributed trees (like P-Grid [53]), and many more. The solution chosen in any system represents a trade-off between the characteristics of these with respect to properties such as durability of data, protection of privacy, system overheads, complexity, semantics of the operations, etc.

2.8.1 Chord

Chord is an implementation of an overlay network and distributed hash table on peer-to-peer networks by Stoica et al. [54]. It can be considered efficient because each node only has to maintain state information about $O(\log N)$ other nodes and lookups can be performed with $O(\log N)$ messages in an N-node system. It also provides that a negative query response means that the key with high probability is not mapped to any value. Chord does not protect the anonymity of the participating nodes nor does it attempt to take advantage of the network topology.

Chord is implemented in a library that is linked with the application and will thus run within the same process at run-time. There are main forms of communication between the library and the application, first a `lookup(key)` function through which the application can gain the IP address of the computer that is responsible for the key. Second, the library provides call-backs to the application for when the set of keys the node is responsible for changes.

The implementation works by arranging the key space in a circle (they called it a Chord circle), see Figure 1. Each node in the network is assigned a value from the key space as its address. It is then responsible for all key-value mappings with key values on the circle sector from and including its address to and excluding the closest address counter-clockwise on the circle, e.g. node 7 is also responsible for keys 5 and 6. This way there will be exactly one node responsible for every key.

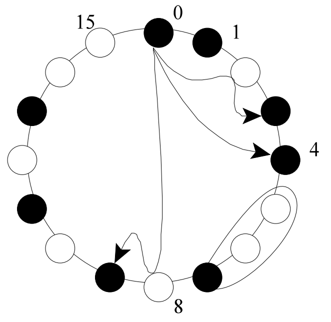

Figure 1 A 16-node Chord circle

Lookups use a so-called finger table containing pointers to nodes in the network so that a query always can be forwarded to a node at least half way closer to its destination. By choosing the nodes in the finger table well it can be kept small, in the order log *N*. In the example ring node 0 wants to have fingers pointing to nodes 8, 4, 2 and 1. When these are missing the finger instead points to the next successor.

2.8.2 Distributed K-ary Search

Distributed K-ary Search (DKS) by Onana et al. [55]. is another implementation of a distributed hash table It is to some extent based on the same ideas as Chord, reasoning about the identifier space in a similar way. A principal difference with DKS is that it uses lazy stabilization of its routing tables, only updating them when they are used rather than regularly checking them. They figured that would be sufficient for efficient message routing and more efficient due to reduced traffic in a network with many more queries than joins and leaves.

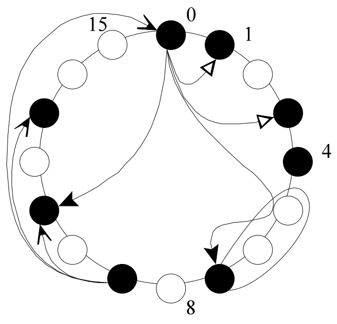

Figure 2 A DKS(16, 3, 2) ring

DKS networks are characterized primarily by three arguments, i.e. DKS(*N*, *k*, *f*). Let’s consider the example in Figure 2. *N* is then the maximum number of nodes in the system, i.e. filled or hollow nodes on the boarder. *k* is the branching-factor which determines the number of fingers on each level. In the example there are two fingers on the first level (filled arrow) and two on the second level (hollow arrow). Note that this is sufficient to always reduce the remaining search-space to one third of the starting search space. *f* is the number of nodes that can fail within a small period of time without breaking the network, thus setting the lower bound on the number of next pointers a node must have (open arrows).

DKS networks are thus in some sense a generalization of Chord networks, since the branching-factor can be customized while it is fixed in Chord. The complexity of the state space to maintain as well as the message count of a lookup is still *O*(log *N*), like in Chord, but with customizable constants.

2.8.3 Tapestry

Tapestry, by Zhao et al. [4], is also an overlay network built on assigning an identifier to each of its nodes (the address) and distributing responsibility for the entire identifier space among the nodes. It then provides primitives for publishing and unpublishing objects in the network as well as routing messages to a node responsible for some object or simply to some node. It thus targets Decentralized Object Location and Routing (DOLR) rather than DHTs as its higher level service.

Tapestry uses a different network abstraction than for instance Chord and DKS. Instead of imagining its nodes arranged in a ring it goes straight to a graph representation. The routing tables are then constructed in levels with address prefixes equal to increasing lengths. This is done in a way that strives to minimize the distance between neighbouring nodes. When a node then is to route a message, it tries to send the message to a node with a longer address prefix equal to the destination address. As already mentioned, messages can be addressed both to nodes and to objects and objects don't have to be located at the node responsible for them. Thus, they also have a mechanism for spreading information about where an object actually is located which also is used to optimize message routing. Messages are routed towards the responsible node in the absence of such information.

2.9 Summary

Our intention was to create a distributed collection framework and thus we have studied several different types of collection frameworks. We have studied local collection frameworks in Java and C++, tuple space frameworks, object databases and finally distributed hash tables. The least common denominator between all of these approaches to collection frameworks is that the collections contain some kind of more or less complex objects.

We also wanted to provide a framework in which it would be possible to explicitly control the coherence and consistency of the data objects, which implied that we needed a significantly complex object layer to build on. For that reason we studied several approaches to distributed object memories. We also studied several approaches to page-based software distributed shared memory, which is today a mature and well understood field but which is also the field where several of the, to us, most relevant consistency models were developed together with implementing consistency protocols.

3 Approach

We set out to design a distributed collection framework. Our first observation was that it should be done in an object-oriented setting, both since that has been the design of choice for other modern collection frameworks and since distributed object memory has shown competitive performance results in other works. To get an object-oriented environment with other proven distributed computing facilities and minimal effort we chose the Java environment. We also expected the choice of the Java environment to facilitate bridging some differences between different computer architectures that could occur in a peer-to-peer network.

We wanted to base our framework on an existing peer-to-peer overlay network to not draw too much attention from the collections. Aiming for high performance, we felt that it would not be right to opt for a technology with somewhat higher privacy and security at the expense of performance. We also thought that having very few false lookup misses would be important in a distributed object system, since these would appear to the program as objects randomly disappearing. Thus we thought that the most promising technology was distributed hash tables. We settled on the Java Tapestry system developed at the University of California, though we believe that almost any structured overlay network exposing message passing and some basic routing information would have been sufficient.

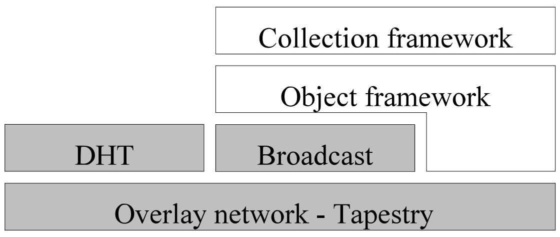

Figure 3 Block diagram on the framework

What we came up with can on a very high level be seen on the block diagram in Figure 3. Our parts have a white background, while other parts have been given a shaded background. As expected the overlay network component is at the bottom and above it are both our services as well as other services.

Figure 4 shows this architecture geographically. Connecting things together is the overlay network with nodes, sometimes referred to as containers, and routes between them through the network. The exact structure of the routes is not important to us as long as the overlay manages to route messages where we want them. On each container there will be local representatives of the global objects. In the figure there are three local representatives (in light gray) which could represent a single global object on their respective nodes. The local representative has some internal structure discussed in Section 3.1. A live network would probably have many local representatives of different global objects on every node.

The Collection Framework does not add new parts to this architecture, but is an implementation of it where the global objects are collections, iterators, etc. This is exactly like the Java Collections Framework which uses Java’s built-in object framework and implements collections in terms of it. The major difference is that our framework builds on a distributed object framework so that the collections can be accessed from any node while the Java Collections Framework builds on a local object framework.

Building on a structured peer-to-peer overlay network, in this case from the distributed hash tables field, provided some crucial advantages. First, it provides means and is optimized for routing messages between any two nodes with approximately the same latency independent of location. Second, it provides means out-of-the-box to connect the network nodes to each other and organize them in a network. Thus we could build a framework where any node could be used as an access-point and participate on roughly equals terms. Targeting distributed applications, all of which has information exchange as its goal on some fundamental level, we considered that a major advantage.

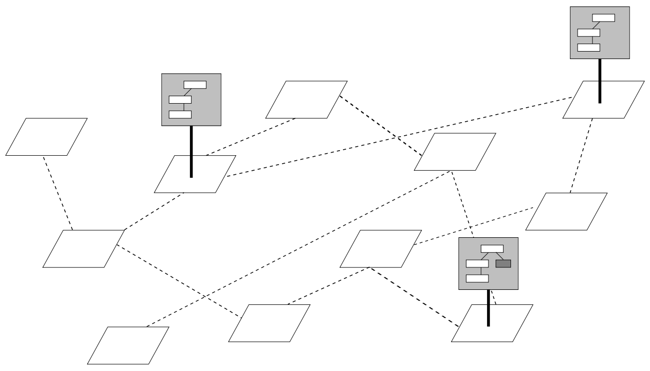
Figure 4 A Global Object on three nodes in a network.

3.1 Distributed Objects

On top of the Java overlay network we needed to develop a distributed object system. What we got, and could use, for free was a system capable of assigning home nodes to objects and routing messages to them, as well as broad- and multicasting within the network. Similar to in DHTs the system must provide for storing and retrieving objects. We also needed to add mechanisms to maintain the appearance that the set of local objects on different nodes resulting from retrieving a distributed object really just provides a view of the distributed object.

The only point of creating distributed objects is of course to use those objects in applications running on top of the peer-to-peer network. This presented a choice on whether if the framework should be constructed to allow easily creating new networks, one for each application, possibly even one for each execution of the application, or as a general network into which applications are "plugged-in" and executed. The second approach appears nice from the perspective that nodes can join the network and share their resources without further due, it would be easier to be altruistic. The first approach on the other hand requires some external procedure for each application to be installed on every node wishing to participate, starting it locally when it is time to run the application, stopping it when done, etc. It puts more of the administrative burdens on the user.

We chose the first approach, again to reduce the scale of our undertaking, while conscious that it would limit the applicability of the framework. Still, it is intended to be a prototype and this decision could be altered in future work. Thus we assumed that every participating node in the network is doing so with the expressed intention of helping performing the specific calculation and that a sufficient part of the program image is available on it.

3.1.1 Object model

We noticed that the Globe [3] project has demonstrated an appealing distributed object model well suited for the Java environment. It makes a difference between plain objects and globally accessible objects which allows separating object-specific logic from distribution mechanisms and policies. Their opinion was that these are both important concerns but that they can and should be handled separately, and this should also suit our goal of explicit control over coherence and consistency.

There is a diagram over our object model in Figure 5. It is very similar to Globe's model, from which it was borrowed. Two network nodes are shown in the diagram and on each there are three parts: The GlobalObjectService (at the top), a *local representative* of a *global object* (in the middle) and some running application (at the bottom). A global object, i.e. a distributed shared object, is made up of all its local representatives on all nodes. The global object is addressed by a unique identifier, its address, and does not need to be represented on more than one node.

The local representative is in turn made up of four sub-objects, the control object, the semantic object, the replication object and the networking object. The control and semantics objects are more specific to the global object's *class*, since these will often need to be written specifically for a kind of global objects, while the replication and networking objects are more general in their nature.

The control object is the local representative's interface to the application and provides the replication object with both information about method calls and means to invoke methods on the semantic object.

The semantic object defines the semantics of the global object, i.e. which data it contains and what its methods does. It is optional in the sense that not all local representatives must contain a semantic object. Although not the general rule, there might be situations where the control object doesn't publish the semantic object's methods directly.

The replication object is responsible for managing distribution and keeping the local representatives of the global object coherent to its stated consistency model. It has a standard interface so that generic replication objects can be developed and reused for new global objects.

The networking object also has a standard interface and is intended to provide a generic network programming model across different underlying networks, given that networking objects are developed for each network.

In practical terms we wanted the process to create a distributed object to be like this: First the programmer writes an object definition in Java. From this he then generates the remote interface (MyObject), the control object (MyObjectImpl) and the semantics object (MyObjectSem) using a system provided tool.

As can be seen in the diagram, the semantic object is not limited to primitive data content but rather consists of the transitive closure of objects reachable from it via non-transient links. This approach makes it similar to a persistence root, in the terminology of persistent programming. It does require that the programmer is somewhat careful or it might not be able to provide the consistency model the programmer opted for when choosing the replication object. An example of such a problem is if an object is passed in and added to the semantic object and the application then continues to directly access it, in which case those changes for instance might be lost or not propagated to other nodes when expected.

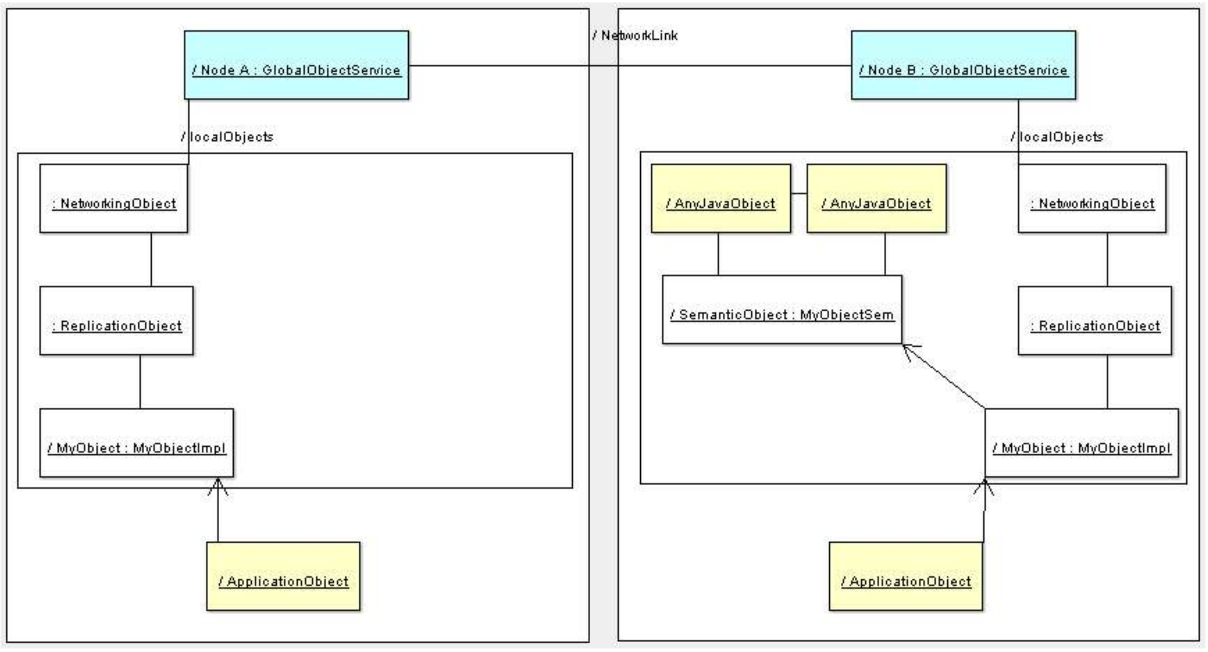

Figure 5 The Distributed Object structure

One might ask why one at all would want a control object. That might seem as an unnecessary indirection for a high-performance solution. We believe however that this separation between the application and the semantic object is necessary to enable experiments with caching using a standard JVM. Since the control object doesn’t contain semantic state for the global object, it can freely be replicated through the system and pointers to it can safely be held by the application. If the application instead had held pointers to the semantic object then something would have to be done to them if the semantic object was invalidated, which would have been difficult. In an updated semantic object would be received then the pointers would need to be updated to point to that object instead, which also would have been difficult. Or, the semantic object would have to be able to create and merge diffs of itself as needed by the framework. Thus we felt that using a control object was a reasonable trade-off between complexity and performance.

3.1.2 The local object repository

An interesting problem is how an object service in a node should hold the local representatives. There are several to some extent contradictory demands on that data structure. When a message arrives one would like fast lookup given an address. When the nodes join or leave the network one would like fast lookup of ranges of objects. If an object is not longer used on the particular node one might want to be able to let the garbage collection reclaim it. On the other hand other nodes might rely on a node to hold some object and then that object must not be garbage collected, at least not on that node.

We believe that a balanced tree structure would be the best way to handle the performance requirements, or perhaps a skip-list (which has similar time characteristics). It would be possible to add a new object, find an object, extract a range of objects and delete an object from that efficiently. These are all operations that can be expected to occur with some frequency. We would guess that finding an object would be the most common operation performed and for that a hashmap might have offered even better performance, although we think that would kill performance for finding ranges of objects. If lookup would become a problem, we would suggest exploring the possibility of parallel tree and hashmap repositories, or a hashed lookup cache.

On the leaves of the tree, we think that there should be some logic. The leaf must be able to hold either a strong or a weak reference to the object, in addition to the address obviously. Thus if the object is weakly-referenced and it is not referenced by the application or some semantic object on that node then it could be garbage collected.

In addition to the structure of the leaves, rules for when an object must be strongly referenced and when a weak reference is sufficient are needed. The first rule is that an object must not be garbage collected on its home node.

Considering a replication policy based on remote procedure calls, we see that the object must not be garbage collected on the node it was created on, since that is the only location at which the semantic object exists. Thus the replication object must be able to prevent the object from getting weakly referenced when the node loses authority over the object, e.g. when it is created or when another node joins the network.

Considering a replication policy instead based on local calls, we see that the replication object must be able to make the object strongly referenced at any time and that it might allow it to get weakly referenced at any time. Any time would be whenever the semantic object shifts location, expected to coincide with a method call. In more detail we believe it would be upon the reception of a replication message, on the receiving end the reception of the semantic object or on the sending end the reception of the acknowledgement of the semantic object. Still, we think that most of the time the reference preference will not change even at the reception of a message.

Thus we think that the leaf should also record the reference preference, i.e. a can-be-weak-referenced-or-not setting, of the object which should be maintained by the replication object. The object manager should provide a facility to manipulate this property.

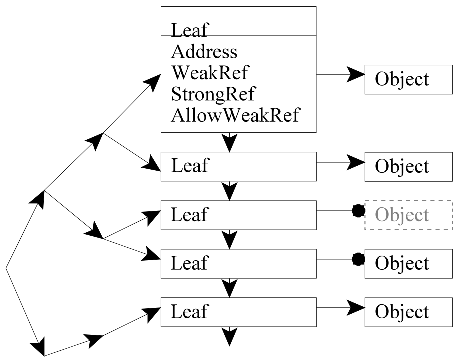

Figure 6 The object manager structure

To conclude we have illustrated this structure in Figure 6. It means to show some part of the tree down to the leaf nodes and one leaf node in detail. The leaf nodes have pointers to global objects, networking objects to be exact, on the right, and the two pointers with circles instead of arrowheads are mean to illustrate weak references. One of the weakly referenced objects is also supposed to have been garbage collected, which will be discovered either a message arrives for that object or perhaps if the object manager periodically checked for it occasionally and removed stale entries.

3.1.3 Method invocation on distributed objects

Method invocation in distributed objects is somewhat more involved than in local objects. Part of the reason why we chose the Globe object model is because the wrapping of the semantics object (containing the class specific code) behind a control object provides an opportunity to manage the coherence of the semantics object before and after the method call, and those are expected to be the only times it might need to be managed (of course noting that this managing can be triggered by invocation on another replica as well).

Methods in distributed objects can be invoked in two ways, either from a co-located user thread or from a network service thread triggered by a network request. In the normal case this is just fine and not particularly difficult, but in the presence of errors some interesting questions surface.

The first kinds of errors that might come to mind are the usual Java `Exception` exceptions and the (perhaps somewhat less usual) Java `Error` exceptions. These should almost always be caused by problems with the implementation of the distributed objects, although other options exist such as the node running out of memory. Regardless they must propagate normally up the call stack of the invoking thread, and if that is remote then it the framework just has to catch the exception and return it as a special return value (functions can return `Exception` instances as ordinary return values, which must not be confused).

Another kind of error is that a node has received a request from another node to invoke some method in some distributed object and then abruptly halts. If it has not begun processing the request then it will be simply like the `invoke` message was lost on the way there and it can safely be invoked on a replica on another node. If processing on the other hand has begun then this might be another matter entirely. It might then have updated other distributed objects or even itself and then those changes might have started to replicate out over the network. If the method now is restarted on another node then the global state might be or become inconsistent.

Unknowingly proceeding in such an inconsistent state can of course be prevented by only ever sending one invocation request, and in the event no completion response has arrived after a suitable amount of time then the state might possibly be inconsistent (at least if the operation is or has invoked a state altering operation (a write)) and that can be signalled.

A piece of often valuable information with an exception in the Java environment is the call stack. In the case of a method invocation on an object located on a different node in our system, that can be seen as the thread leaping to that node leaving the bottom of the stack behind. That forces the thread to leap back when returning from the method invocation, but here it means that only a partial stack will naturally be available when an exception is thrown. We decided to live with that and only add lower stack frames when the exception is transmitted back over the network.

This solution does at the very least allow the programmer to follow the call stack from where the exception was caught to where it was thrown. Compared to ordinary exception handling the exception might not reveal how the program came to where the exception was caught and thus the programmer might have to take steps on his own to identify the place on his printout or in his log. On the other hand the framework might not have any opportunity to catch and modify an exception before the application on regular JVMs, so this appeared to be the most reasonable trade-off between being consistent and practical to the programmer and a vastly more complex programming task.

An entirely different problem is how to specify whether if operations in distributed objects are possibly writes or not. This might be desired by the replication objects to avoid performing unnecessary work to synchronize replicas, for instance after having invoked a method that only reads values in the local replica or even calls writing methods but only on other distributed objects.

Java does not have any native notation for this, which is similar to const member methods in C++. This annotation should be provided by the control object to the replication object. In Globe, the opposite to this was denoted by adding the `_w` suffix to function names. So this was denoted by the lack of the suffix.

Someone familiar with e.g. CORBA would also expect an interface definition to define the directions of arguments to operations, such as in, out or inout. We chose not to add this since it would not be consistent with the RMI mechanism and not really meaningful. Java is a pass-by-value language and thus arguments cannot by themselves transmit output from a method call. Local method calls can do so through state changes on an object pointed to by a parameter, but we thought that repeating this in a distributed environment would pose significant challenges with regards to updating the object on the calling node after the method invocation.

3.1.4 Blocking method invocations

A particular problem occurs when the semantic object also contains references to other distributed objects. Calls to distributed objects might block for long time, which indeed might be the case also for e.g. I/O operations. To some extent it is irrelevant if the thread executing in the distributed object is blocked or if it computes, regardless it is busy. A node in the network would be expected to have some limited number of threads to execute incoming requests, and while these are busy they are obviously not available to service new requests.

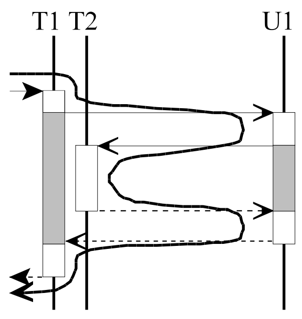

Figure 7 A distributed recursive call.

A node could thus run out of free threads and lose its ability to service requests. If the completion of its outstanding requests then depends on the completion of some of the incoming requests, like could be the case in Figure 7 if T2 were occupied, then deadlock is a risk. Various schemes to increase the number of simultaneous outstanding requests that the system can allow, beyond the number of worker threads, are possible. While raising this breaking point some might still have a breaking point, some might introduce additional failure scenarios and many requires special programming styles.

This problem is not unique to us, but general to the remote procedure call situation. Globe's decision to not allow the thread to block solves the deadlock problem, but at the price of increased complexity to the programmer. The restriction does still not guarantee that all incoming requests can be services in a timely manner, but can to some degree be considered a request to the programmer to avoid a particular kind of potentially lengthy processing. In the end that comes down to the scheduling law that a system that receives more requests per time unit than it can service cannot be scheduled, and the shorter time each request requires the more requests can be serviced.

Having considered this we decided that requesting that programs do not block in their distributed objects is a too extreme measure. That is a policy decision that the programmer should make for himself, conscious of the implications. Thus we decided to keep an as simple design as possible and to not introduce some new mechanism for this reason.

3.1.5 Distributed threads

Distributed threads comes in different shapes and sizes in other systems, e.g. CORBA has the `oneway` asynchronous method call and Linda has active tuples. RMI does not directly support thread creation and certainly not thread mobility. It is possible also in RMI for the called object to spawn a new thread in that JVM and leave all processing to that thread. The caller would have to wait the entire round-trip time so the latency would be higher than for a strictly asynchronous operation, but still probably quite low.

We did not want to go entirely in the direction of RMI, but let threads be some kind of explicit objects in our framework. This was because we, like RMI, didn't intend to impose thread mobility on Java but needed some light-weight concurrent execution facility to enable parallelizing operations, e.g. if a list would be made up of sub-lists and the operation could be carried out on them simultaneously.

To meet this need we decided to use a simple thread-pooled execution service to which tasks could be assigned. It was mainly intended for short-running jobs since when all threads were busy new jobs would have to wait, perhaps forever if jobs depended on each other. This should not be a severe restriction since it would still be possible to start new threads as outlined for RMI and for long-running jobs a slightly higher start-cost should not be a big concern. Our service could even be used to hide some of the start-cost. Thus we felt that this was a reasonable choice, although one that very much relies on our assumption that the network can be trusted.

Similar to synchronous method calls threads can also suffer from that the node holding the bottom of the stack leaves the network. In this case there is no node waiting for a return from a method call that naturally would detect this and throw an exception to the application, and not necessarily any natural thread to throw the exception on nor a well-defined place to throw it that the application could possibly predict.

We decided not to solve that problem. If an application would like to know if and/or how a task terminated then the best way to find out would be location dependent and so the application should choose. If the task is co-located then it could for example add the result to a collection of completed results, otherwise it could for instance call some global object with the result.

3.1.6 Object Framework API

We have included some parts of the javadoc from the most important classes belonging to the object framework from our prototype in Appendix C. We will just discuss it very briefly here.

`GlobalObject` is the interface for control objects. Missing in the prototype were a `void delete() throws RemoteException` function, as well as `void setDoFast()`. We would probably exclude `boolean getDoLocal()` if we would do this again, just as we would avoid adding a `boolean getDoFast()`. `equals()` and `hashCode()` are listed only to point out that they might throw the unchecked `UnsupportedOperationException` exception, which would be unexpected by anyone only having read their original specification (see further in Section 4.2).

`NetworkingObject` it the interface for networking objects. As we say in Section 4.2, maybe this object could have been excluded altogether. `P2PReplicationObject` is the implementation of this interface in our prototype.

`ReplicationMessage` is an interface that message classes addresses to replication objects must implement. We believe it might be worthwhile to add some `int getMessageId()` in an implementation that aims to handle errors properly.

`ReplicationObject` is an interface for replication objects. The `created()` function comes in two versions, one for getting a random address for the object and one for supplying one. We think that the version without argument should perhaps have had a `boolean` argument for determining if the address should be selected from the total address space or the local address space. One of them should be called at the end of the control object's constructor (see Section 3.6 Figure 15). Missing here also are the `void delete() throws RemoteException` (see Section 3.4.2), and `void setAtHome(boolean)` functions. The latter could be an up-call for when the node becomes or stops being the home node for the object, in order to manage the reference policy discussed in Section 3.1.2. `RPCReplicationObject` is the implementation class of this interface supplied with our prototype, see also Section 3.5.1.

`InvocationResult` is the class that is used to hold the return value of an invocation. Control objects shall use it to return the result in their `invoke` method (see Section 3.6 Figure 16), and might thus need to unpack it in their method wrappers (see Section 3.6 Figure 17). Doing so also for `void` methods can detect a class version inconsistency problem. While we believe it is sufficient for most cases, it should be extended to support all primitive types.

`ObjectManager` is the central class of the object framework and the centre around which everything else revolves. It is a singleton. Global objects must register their networking objects with this class since it can multiplex incoming replication messages much more effectively than if the networking objects had registered directly with the overlay network. It also provides access to some additional services, such as resolving global objects by name (see Section 3.4.3) and running jobs (see Section 3.1.5).

3.2 Distributed Collections

Once the distributed object framework was sufficiently worked out, we intended to build our distributed collection framework on top of it. We needed the object framework for that for two reasons, first the collections were going to be distributed objects, and second the internal data structures of the collections would be built of distributed objects. This way we would not just have collections accessible from many nodes, but indeed we could make collections that could be divided over many nodes and thus for instance much larger than any single node could hold.

Java already has a well known and appreciated collection framework for single-processors and tightly coupled multiprocessors, so we decided to try to make our collection classes support as similar interfaces as possible. So decided to port the Sun interfaces to our object framework. Then we would create global objects wrapping classes from Sun's collection framework such that they implemented the new interfaces.

One thing to consider about global collections is the objects that one might like to put in them. Obviously they need to be serializable. One should probably generally avoid putting very large objects or object structures in there, in the local case it doesn't matter since then it is just pointer programming but here the objects might need to be serialized and that is not pointer programming. A notable special case of this is putting global objects in collections, which we would generally advise against. A global object would then be registered with the object manager not only on the putting and on the getting node, but also on the storing node.

Thus we would recommend that the address of global objects be put instead and then the object resolved on the getting node as the general rule.

Another thing to consider is if the collection framework should be largely stand-alone and client applications contain only a small interface to interact with it. Alternatively the collection framework could be completely contained (embedded) within the client application. We chose the embedded solution since we then could use low-latency intra-application communication between the framework and the application. Then the overlay network was also directly available to the application, we could avoid writing an extra communication layer between the application and the framework and not least because an embedded framework could be used to construct a stand-alone framework simply by adding the application communication on top of it.

This choice was not without negative consequences however, the application has to contain the framework service threads, leave enough file descriptors for the framework, the framework will be less protected from the application and vice versa, resource consumption by the framework (e.g. for RPC) will be charged to the application and so on. We chose the solution that we expected the highest performance from.

3.2.1 Wrapped collections

We mentioned above that we intended to wrap a number of classes from Sun's collection framework in our framework. This meant more specifically that we would create control objects that would use Sun's classes as semantic objects, the remote procedure call replication policy and the peer-to-peer networking object. The idea behind this was to quickly get some real collection classes with our new interfaces. We selected `ArrayList` and `HashMap` to wrap in this way.

Selecting the remote procedure call policy was crucial for wrapping these classes. Consider for example `Iterator` itself or that `List` has no calls to operate on ranges of objects. The `Iterator` instance is tightly tied to the collection, so tight that it is in fact illegal to use if the collection is structurally modified through any access-path not going through the `Iterator`. In the other case the solution is to make a sub-list, which would have much the same properties as those of `Iterator` just discussed.

What the remote procedure call policy did for us was to provide sufficient conditions so that these tight bonds between the semantic objects of different global objects could be managed. The problem that turns up, perhaps with any other replication policy, is that these bonds are not easily serialized. With the remote procedure call policy we are assured that the semantic objects are never serialized and thus that problem is avoided. On the other hand the collection is trapped on the node that it was created on.

This arrangement can be seen in Figure 8. Let the global object on the left be some kind of collection and the object on the right an iterator of some kind over that collection. In the global objects are a control object (white) and sometimes a semantic object (yellow). Then there would be an actual reference from the iterator to the collection semantic objects, denoted by an arrow on the diagram (in practice this attribute might be hidden as part of an inner class, but it is there nonetheless). Node B can use remote procedure calls to access the semantic objects as long as this relation

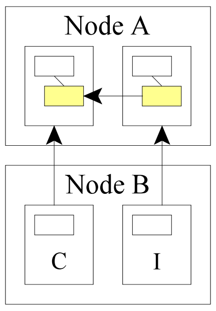

Figure 8 Wrapped objects on two nodes

remains intact, and that is precisely what would be difficult with object relocation. This example would look the same for sub-lists, sub-sets etc.

This also means that a collection wrapper must be have access to general wrappers that can encapsulate the result from e.g. a call to `iterator`. Obviously it would not be okay to copy the collection iterated by the iterator in this case, but it must really become the semantic object of the new global object. There would not be very much to this, as expected, but it was necessary.

3.2.2 GlobalAggregateList

We have mentioned only a few times so far in this report that we wanted to construct collections spanning multiple nodes. This is the one that we eventually implemented, and it uses the wrapped collections from the previous sub-section heavily.

We have illustrated the basic idea of `GlobalAggregateList` in Figure 9. The aggregate list is marked with fat lines and its semantic object resides on node C. It aggregates three `GlobalArrayList` lists with their semantic objects on nodes D, E and F, let's assume in that order. This means that A and B can access the aggregate list through the local representatives of it on those nodes. These accesses will go through the semantic object on C, which is basically a list of sub-lists, to the `GlobalArrayList` sub-list's local representative on C, and in the end access the elements in some or all of the lists on D, E and F. It also means that the indices of the elements on D are lower than those of the elements on E, which in turn are lower than those of the elements on F.

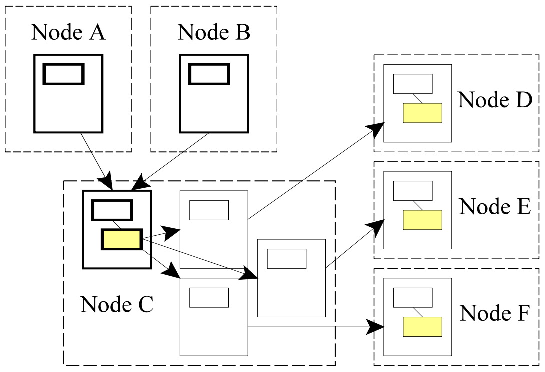

Figure 9 GlobalAggregateList arrangement

Such accesses would naturally follow the normal paths through the diagram. They would start at the local representative of the aggregate list on A (or B). There they would be directed through the semantic object on C and on to the local representative of the list on D (or E, or F, or some combination). Finally they would reach the semantic object of the list on D and the result would then bubble back the same way but in opposite direction.

It is pretty clear now that several operations should be possible to execute in parallel rather than serially and thereby possibly hide some call times (be it network overhead, processing time or whatever). This is most obvious for operations that apply equally to all sub-lists, i.e. `clear`, `removeAll`, `retainAll` and `toArray`. `contains` could also do this and would not even need to wait for a reply from every sub-list if a positive response was received. A caution with regards to that is that a subsequent destructive operation on the list could then get to execute concurrently on some sub-list, which could lead to errors.

`indexOf` and `lastIndexOf` are to large extent similar to `contains`. They could also, with the same caution, return early but only on condition that they have received negative responses from all preceding respective succeeding sub-lists. Without this condition they would not necessarily find the first respective the last occurrence, although they would find some occurrence. `remove(Object)` should qualify near this group, it would quite possibly be good to implement it using `indexOf`. One should probably not lightly try the optimistic distribution of remove requests though, as only a single occurrence may be removed and fixing things up if it went wrong would be difficult.

A few operations, `isEmpty` and `size`, don't even need to involve the sub-lists but could be answered directly by the aggregate list. We think that it would be worthwhile for the aggregate list to cache the sizes of the sub-lists. That would be sufficient to answer these calls, but would also be immensely useful for calls like `get`, `set`, `add(int, Object)` etc.

A function for which we have found no efficient design is `containsAll`. Obviously it would be insufficient to ask if any sub-list contained all elements and the answer would not reveal which elements are left. We thus simply iterate over the collection and check if the aggregate list contains all elements.

In this case it would not be possible to wrap some iterator object the same way as were possible with wrapped collections. Instead we settled for making an iterator that counted position and interacted with the list by index. This is less safe than local iterators (which also are unsafe) with respect to concurrent modifications to the list and the iterator. The aggregate list is however pretty efficient with indices, so it will probably not be a big performance issue.

We left `equals`, `hashCode` and `subList` unimplemented. The first two for reasons discussed in Section 4.2 and the latter since creating such lists unaligned with the sub-list boundaries would be a bit messy and not required for our evaluation. At that time we also had doubts about the viability of that design pattern, compared to for instance index or iterator restricted operations.

3.2.3 HashSet and HashMap

We also considered a `HashSet`/`HashMap` design not conforming to the Java Collections Framework. In this design we restricted the objects that could be contained to only global objects. Neither did we include support for iteration nor sub-sets. It was intended to dress up the functionality of the underlaying overlay network in a collection like way and offer few but efficient functions.

First we noticed that `HashSet` and `HashMap` in many ways are very similar collections. `HashSet` uses a value to look for the object in a place where there with any luck are few other and verifies that it is there. `HashMap` uses the value the same way and returns an object pointer instead of a true if the object is found. For this reason we considered system support for HashMaps sufficient, since the difference could be hid by the front end class.

Our intention was to solve these in the most straightforward way, by letting `HashSet` and `HashMap` be distributed objects. Thus each collection instance has a stable identifier that we will use to tag distributed objects acting as keys in a map together with the identifier of the value object. In a set the object would be both key and value, thus tagged with the set and its own identifier. This interaction with a set in mind is shown in Figure 10 as the message sequence starting with 1, the difference for a map would be an additional argument to put in message 1 and that argument used as the second parameter in message 1.1.

The expected time complexity would be $O(\log N)$, since that is the number of messages required by the overlay network to route the tag message from Node to OwnerNode and the other operations can be implemented in $O(1)$.

The interaction for looking up the value object of some key object (testing if an object is in a set) is shown in the message sequence starting with 2. The id parameter could either be a reference object or a distributed object that the map then would take the reference from. The interesting thing to see happens at message 2.1.1.1.2 if OwnerNode is not responsible for the value object. OwnerNode then injects an object request into the network addressed from Node and forwards it by normal means. OwnerNode also sends a redirect back to Node so that Node can identify a push of the value object as the response to the query. This is also useful for fault management as Node then can see that the request is progressing and can restart the request closer to the target if for instance a message would be lost.

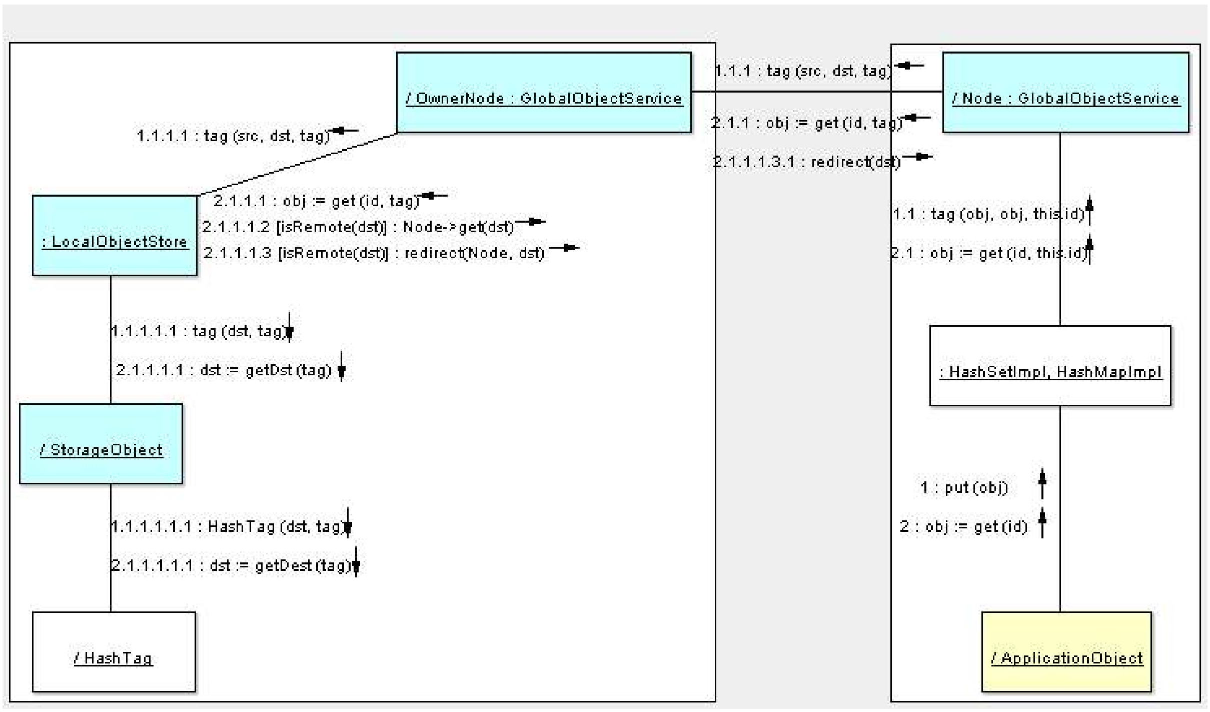
/ OwnerNode : GlobalObjectService
1.1.1 : tag (src, dst, tag)
/ Node : GlobalObjectService
2.1.1 : obj := get (id, tag)
2.1.1.1.3.1 : redirect(dst)
1.1.1.1 : tag (src, dst, tag)
2.1.1.1 : obj := get (id, tag)
2.1.1.1.2 [isRemote(dst)] : Node->get(dst)
2.1.1.1.3 [isRemote(dst)] : redirect(Node, dst)
: LocalObjectStore
1.1 : tag (obj, obj, this.id)
2.1 : obj := get (id, this.id)
1.1.1.1.1 : tag (dst, tag)
2.1.1.1.1 : dst := getDst (tag)
: HashSetImpl, HashMapImpl
/ StorageObject
1 : put (obj)
2 : obj := get (id)
1.1.1.1.1.1 : HashTag (dst, tag)
2.1.1.1.1.1 : dst := getDest (tag)
/ HashTag
/ ApplicationObject

Figure 10 HashSet and HashMap insertion (1.*) and querying (2.*)

3.2.4 Collections Framework API

We have included some parts of the javadoc, from the most important classes belonging to the collection framework in our prototype, in Appendix C. We will just discuss it very briefly here (one might want to skip forward to Section 4.3 and peek at Figure 20 too).

`GlobalCollection` is a direct parallel to `java.util.Collection`, with the addition that it extends `GlobalObject` and thus should be implemented by the control object of the concrete global collection class. On a collection that is not known to be co-located (by whatever means, e.g. a replication policy that guarantees it) `toArray(Object [])` should always be avoided. It might seem tempting to add methods for managing an internal set of iterators, thus being able to create something like a generic iterator semantic object whose data is a `GlobalCollection` pointer and an iterator handle (and the semantic object entanglement discussed in Section 4.3 would have been solved). We would rather see such functions in an orthogonal interface since we believe that would be sufficient and would leave the ultimate decision to the designer of the specific collection.

Another issue to consider is that this API specifies that an iterator is (potentially) invalid whenever the iterated collection is structurally modified, unless it was modified through that particular iterator. This could appear practical enough on a tightly coupled system, but we feel some concern that on a more loosely coupled system this will be impractical. It could for instance prevent different nodes from using several iterators to work on different parts of a list, or require complex synchronization between them to invalidate the iterators at safe points and then obtain new iterators. In this case `GlobalDecoupledListIterator` would not help much either, even though it doesn't start to throw `ConcurrentModificationException` exceptions. It does however not move with the changes, so whenever a node working further down the list adds or removes an entry all iterators at higher indices appear to move. This argument goes also for other entangled objects such as sub-lists, key-sets, value-sets etc. and also for other collections than list (which was chosen as example since it is pretty obvious how related objects could be affected).

`GlobalList` is a direct parallel to `java.util.List`, the same way as `GlobalCollection` (which it also extends). This interface defines more methods potentially returning entangled objects, specifically `listIterator` and `subList`. With respect to `subList` it might be worth examining further if and how it is used in practice and if simply replacing it with `remove(from, to)` would not be sufficient (thus covering the example use `list.subList(from, to).clear()` from the API).

The main differences between `GlobalList` and `List` are otherwise the `equals` and `hashCode` functions, which have an extended specification for `List` classes compared to classes in general. We believe that these should not at all be a part of the collection API. We expect distributed collections to be larger than usual and that distribution issue must be considered when methods like them are implemented or they could be exceptionally slow.

`GlobalMap` is a parallel top `java.util.Map`. We removed the `entrySet` from the interface since maintaining its semantics would simply have been way too expensive, and it is redundant. Also here there are opportunities for entangled objects, specifically `keySet` and `values`. One might consider adding a `void putAll(GlobalMap)`. An alternative is to use the Java trick of embedding the `GlobalMap` within a serializable class, so it can be passed as parameter, implementing the `Map` interface, that would rewrite any `RemoteException` exception into some unchecked exception. Here too `hashCode` and `equals` are a problem.

`GlobalSet` is a parallel to `java.util.Set`. It has much the same issues as `GlobalList` and `GlobalMap`, so there is not much more to add about it.

`GlobalIterator`, `GlobalListIterator` are obviously parallels to `java.util.Iterator` and `java.util.ListIterator`, and we don't have much more to say than has already been said about them either.

`GlobalAggregateList` is our show-case implementation of `GlobalList` (see also Section 3.2.2). It has the append operation that allows a clone of a `GlobalArrayList` to be added to the end of the list (see further discussion in Section 4.3). Inside its semantic object there is a list of sub-lists that reference the clones of the lists that has been appended. Missing from our implementation is the `subList` call. For this class there is also no native iterator, since we have written the entire semantic object from scratch. So for iterators we implemented `GlobalDecoupledListIterator`, which as the name implies has a semantic object that is entirely decoupled from the semantic object of the list. It uses a counter to track its steps through the list, uses index based operation on the list, and thus doesn't feel if the list changes "concurrently" unless it would suddenly find itself outside the list.

`GlobalArrayList` is our wrapper class (see Section 3.2.1) around `java.util.ArrayList` and implements `GlobalList`. As such it becomes an entirely monolithic collection, i.e. one that is confined to a single node. It uses the RPC replication policy, which is a requirement for it to work properly as it uses `GlobalFakeIterator` and `GlobalFakeList` to create iterators and sub-lists. Notable is also that it is cloneable and that the clone is created on the *server* node, i.e. the semantic objects of both lists will be co-located, and not on the *client* node, where the `clone` call was made from. We borrowed the unchecked `java.rmi.server.ServerCloneException` to box `RemoteException` exceptions during the cloning.

`GlobalHashMap` is our wrapper class (see Section 3.2.1) around `java.util.HashMap` and implements `GlobalMap`. Obviously it doesn't support `entrySet`. It is also monolithic and uses the RPC policy, which is good since it relies on `GlobalFakeCollection` and `GlobalFakeSet`.

An instance of `GlobalHashMap` with the well-known id 4711 is used by the `Naming` class in `se.kth.p2p.util` (not included in the appendix), which should be created by the root node when it creates the network. In the future we hope that the collection could be replaced by an actually distributed collection.

3.3 Network operations

As already mentioned we used the Tapestry overlay network to pass messages between the collaborating nodes. This meant that the network operations available to us were highly constrained by the way Tapestry worked. That was not always quite as clear as we could have desired.

3.3.1 Creating a network

Creating a network or starting an application might not be a terribly difficult thing to do. The first node must initialize the overlay network the way that would normally be done with the particular overlay client used. To begin with it will have to be responsible for all object identifiers, and then obviously store all objects and carry out all computation locally unless other nodes join. Exactly how the work is split among the nodes or the nodes synchronize we considered application specific.

A particular task that should be handled by the first node is the creation of well-known objects. For example we have a simple naming service, for convenience, and it has a well-known address so that all nodes can resolve it.

The actual configuration of Tapestry, finding a node to connect to (relevant when joining) and so forth is primarily up to the user. This should be done through an XML-like configuration file fed into Seda. We have provided a stage that can initialize our framework and start a `Runnable` user class, again for convenience. It is perfectly feasible to initialize our framework on your own.

3.3.2 Joining a network

Joining a network is a more complex operation than creating it. The overlay network client probably make some adjustments to its routing tables and we will see changes in which node is responsible for which keys. Responsibilities for distributed objects might thus shift. A network hoping to provide some fault tolerance might also need to maintain the number and location of replicas.

What will happen for us is that Tapestry will connect and integrate the node with the network and then we will see a key responsibility gain on the joining node and corresponding key responsibility losses on the nodes already in the network. The nodes previously responsible for those addresses must then offer them to the new node so that they can continue to be resolved.

Care should be taken about the time between the new node joins and that it is up to date on all objects that belong to it. A resolve in this time-window could otherwise be falsely rejected, and at the same time the window must be closed so that resolves of actually non-existing objects are eventually turned down. We suggest that nodes just wait a while before rejecting queries when they have just connected.

3.3.3 Leaving a network

Yet slightly more complex we have the operation to voluntarily leave the network. In this case will perhaps eventually want to update routing tables, but before that the object framework would want to hand of objects to other nodes in the network. This would include the control object, but the replication object must be informed such that the semantic object has a chance to survive. Additionally a fault tolerant network is very likely to need to create new replicas on other nodes now, which is also a reason to involve the replication object.

Because of the difficulties involved in moving live threads and active method invocations mentioned previously the node should wait for them to complete before actually leaving the network. Obviously it should stop starting new method invocations when it starts to leave, and either immediately defer them to other nodes or queue and hand them over to the new responsible node. To possibly make this reasonable there should be a flag that long running well-behaved threads and method invocations can query occasionally and if set try to clean up and terminate/return rather quickly. This all requires that Tapestry has not yet left the network since then it would seem difficult to communicate with other nodes about these things.

In the most extreme case the node is the last node on the network, and the network will cease to exist after the node has left it. This is however perhaps the simplest case, in the sense that then the distributed application terminates and all information can be discarded.

3.3.4 Message passing

Message passing between nodes in the network is obviously also an important operation since it is the foundation for all higher level operations. It is to a large extent provided by the overlay network client and we would have to do little more than to process and respond to incoming messages, at least on this level.

3.4 Object operations

Object operations are higher-level operations built on top of message passing. These make up the foundation of the object service and define the ways that all objects can be manipulated, e.g. how to create, destroy or resolve a global object. They do not play any role in any consistency protocol, that is the role of replication operations covered in Section 3.5.

3.4.1 Create object

Creating an object is a pretty straightforward thing to do. First the peer should create the distributed object structure locally, selecting a replication object and a networking object.Then the distributed object should be given a global name/identifier. This name should be unique and within the address-space of the overlay network so that it will have a well-defined home node.

The name can be selected in different ways, e.g. selecting a free name among those that the local node is responsible for, through testing random names until a free name is found, etc. Selecting a local name means that the object will be homed on a node that at least once accessed it, which might or might not be an advantage, but also that distribution of objects is likely to get skewed in networks where few nodes create many objects.

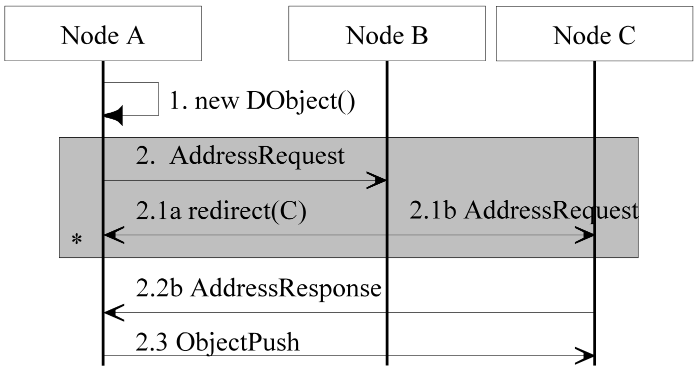

Figure 11 Object creation

We thought that avoiding this clustering would be the better choice and thus decided to use a combination of the two schemes. In this scheme, shown in Figure 11, an `AddressRequest` is sent to a random node. The node should then respond with any free address it is responsible for, alternatively redirect the request to a new random node. When node A has received an `AddressResponse` containing the address for the new global object then the so far anonymous object is named and pushed to its home node so that it can be resolved by other nodes. In the mean time node C has reserved the name so that it will not be given to several object and incoming resolves for that object can be stalled until the object has been pushed to C.

If C thinks the push has taken too long it can try sending an `ObjectRequest` to A. If this succeeds then the object has been pushed, and if not then the address has somehow been lost and can for instance be treated like the address of a deleted object.

We didn’t expect many collisions to occur, nor the address-space to run out, during naming since the address-spaces of most overlay networks are gigantic and we though only a small fraction of it would be occupied at any given time. Should the control structure of a global object be deserialized and there exists a conflicting object there then it could fail to deserialize with an exception to indicate the problem. Should a node still run out of address-space then we believe it could just redirect the request to another node and expect better luck.

There is also a legitimate need for objects with well-known addresses, for instance it would be difficult to make a naming service without that - how would the name to object map be located? One solution would be to for instance map the positive `int` values to some range of addresses and make it possible for the user to give the address the an object should be given.

It should be recommended that such named objects be created as early as possible, perhaps even before any other objects, to minimize the risk that some other object by chance has taken the requested address. Perhaps it would be sufficient only to create such objects on the node responsible for the given address, which would boil down to that they should preferably be created before any other objects on the root node before any other node has connected.

3.4.2 Delete object

When objects can be created then there need also be some mechanism to delete them. Two alternatives for that are to provide an explicit `delete` primitive or to use garbage collection. Garbage collection can and has been implemented in distributed environments, some that even can handle nodes being off-line at times. We did however feel that a `delete` primitive would be a simpler and sufficient solution.

Performing the actual deletion could be a little tricky since the replication policy might try to rescue the object if it doesn't know it is being deleted on purpose. One possibility is to make that difficult for the replication policy by broadcasting the `delete` message and pray that object doesn't move from a node that the broadcast hasn't visited to a node it has already visited, in which case it would at least partially survive. Another possibility is to accept that it could partially survive for a while but delete it on enough nodes to make sure it stays deleted.

We believe that the latter alternative is the only one that is reasonably unlikely to fail to delete the object. Since the replication object is the object that handles replication issues we found that there are compelling reasons for the object framework to rely on it to delete the object.

Like in other environments with explicit delete primitives it might happen that a deleted object is used after having been deleted. Just that the replication objects has coordinated the deletion doesn't mean that all references to the global object have been dropped. We do not intend to specify the result of using such a reference, but note that an exception would probably be expected by most programmers.

3.4.3 Resolve object reference

Resolving an object reference is a very nice little operation, shown in Figure 12. All that needs to be done is to send an `ObjectRequest` to the home node and wait for the `ObjectPush`, and the request could be repeated if needed. Obviously it could happen that there is no object with the requested address and then the request should be rejected and an exception generated. Should no response at all have been received after some time then the requesting node could optionally send a new request.

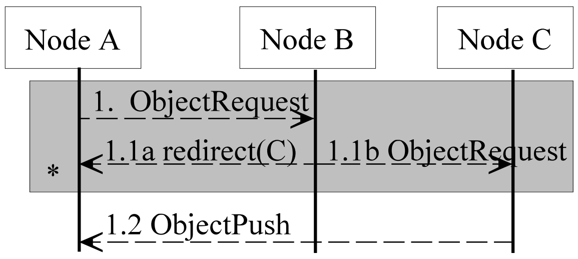

Figure 12 Resolving a reference

3.4.4 Putting objects

We have sketched on two ways for informing a node about a global object. They are not entirely different but rather one is a part of the other. Our idea is that the extended variant could reduce processing costs but have a higher latency, and thus that the shorter variant is appropriate in situations where latency is critical or it would be difficult to apply the extended variant.

The extended version, see Figure 13, starts with node A sending an `ObjectOffer` containing the addresses of some objects to B. B then checks which objects it is already in possession of and which objects it actually wants and sends an `ObjectRequest` with the addresses of those objects. A should then respond by sending those objects in an `ObjectPush` message. This extended sequence can for instance be observed while a node is connecting if there are global objects for which the new node will be responsible.

The `ObjectPush` message can be very simple and just contain references to the control object or networking object of the requested global objects. Because of the interdependencies in the local representative the entire local representative, except the semantic object, will be serialized when either object is serialized. The objects are then required to register themselves with the local object manager on the node they are deserialized. It might happen that the object is already present on that node and then the deserialization process must replace the deserialized objects with the local representative already present.

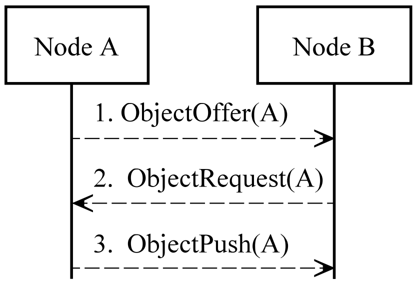

Figure 13 The extended global object push sequence.

It might at first seem unnatural that the object should register itself, but this brings us over to another way that objects travel between nodes. A reference to a global object can for instance be passed as parameter to a remote procedure call, be contained in local objects passed as parameters or even be part of semantic objects sent between nodes. In these cases the local representative would also be serialized, perhaps inside some other objects, and we saw no natural way for any external entity to register the object while it was being deserialized. Letting the object register itself we avoided the need for a different mechanism for these two cases.

3.5 Replication operations

Replication operations also belong to the higher-level operations built on top of message passing. The ones discussed here might not be all operations that will be invoked in a live network as an application has considerable freedom to construct and use new replication protocols through the replication-object mechanism.

We have created one simple replication policy, since a replication policy is an absolutely necessary part of every global object. Due to time constraints we could not proceed with the development of more policies and the implementation of them as replication objects.

3.5.1 Remote procedure call

The first policy that we wanted to work out was a remote procedure call policy. The default policy in RMI is based on remote procedure calls. It might also be the most general of all policies in the sense that it adds no constraints on the semantic object, compared for instance to a policy that moves the semantic object and thus requires it to be serializable. The cost of that is obviously the constraints it adds on the node the semantic object resides on, as well as the constraint that all parameters must be serializable.

The minimal requirement is that there is a message for invoking the method and another message for returning the return value/exception. Additionally an acknowledgement that a return value has been received and a message for polling if

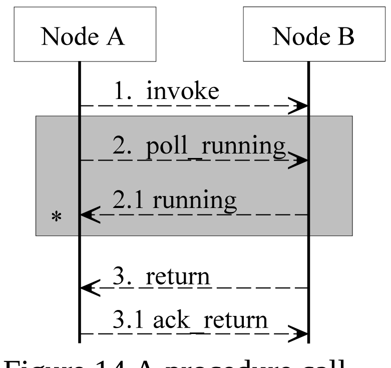

Figure 14 A procedure call

an invocation is still in progress might be desired. This extended sequence is shown in Figure 14. We thought that resending the return value would be safe, so an acknowledgement would be useful.

A variant of the message sequence is that node B receives a poll message after having sent a return message. This might happen if the poll and the return are sent at the same time or if the return has been lost. Another variant is that node B receives a poll message after an `ack_return`, e.g. if the poll has been delayed, and then it should be discarded. Yet another variant is that node B receives a poll without preceding invoke, e.g. if the `invoke` message was sent to another node that then somehow lost contact with the network, and then a `not_running` should be sent back.

A variant on node A is that a `running` or a `not_running` message is received after a return. It should just be discarded in this case since the application has already continued. Another variant on node A that it receives no response to some number of polls or even receives a `not_running`. It should then fire an exception on the blocked thread and stop waiting. Should A receive a `return` message later then A should just acknowledge that to allow the sender to free any resources. Same should be done if A would receive a `return` message without having sent an `invoke` message. This is also not entirely altruistic by node A since it should also like not to receive some number of retransmissions from node B, which would be the case if it didn't respond to the `return` message.

As previously discussed (in Section 3.1.3) it is unsafe to send more than one invocation request, so there is no reason to acknowledge it. An invocation request must contain some data: An invocation request id, the method to call and all arguments. The object to call the method on is implicitly named by the request, it being the destination address.

The policy doesn't constrain the semantic object to be serializable, which should be the case with most other policies. Because of this all method invocations must be executed on the single semantic object that the global object started out with, and thus the remote procedure calls discussed above are necessary. In order to invoke the method on its resident location, i.e. the node where the semantic object is, the parameters must be transferred there and eventually the result transferred back. Thus all parameters and the return values must be serializable or `void`, which is a slightly weaker but sufficient requirement than that the argument types must be serializable themselves. Finally it should be noted that because the resident location is fixed that node must not leave the network until the object has been deleted or the computation terminated, since then the semantic object would no longer be accessible to the system.

3.6 Control objects

The control objects are one of the two parts of a global object that the object programmer necessarily will have to provide. As previously mentioned they are the facades of the global objects to the application. They also play a role when requests arrive over the network and the replication object wants to invoke a method on the semantic object.

Control objects are very regular objects in practice. We believe that generating a control object could generally be done pretty mechanically from the semantic object. We think that a generator reading control object code, or something very close to it, would in fact be a very useful thing that should be a part of a complete environment. We imagine that it would perhaps be difficult to cover all cases with such a generator, but think that it might be sufficiently useful just to cover the case that the semantic object is a simple Java object exporting its public methods.

The first thing that must be done to a global object is to create it, for obvious reasons our scheme would not work well with static data and methods. Thus, the control object should have a constructor. We would suggest a construction like the one in Figure 15. It first creates the objects of the local representative and connects them together. Then, when calling `created`, the object is registered with the object manager and given an address. Optionally arguments could be accepted by the constructor and passed on to the semantic object. Yet another option is to accept a `NetworkAddress` and let the global object to be given a name, in this case `replicationObject.created(NetworkAddress)` should be used. It is also during the registration of the object as the `RemoteException` could occur.

```
public MyGlobalObject() throws RemoteException {
  replicationObject = new RPCReplicationObject(this);
  networkingObject = new P2PNetworkingObject();
  semanticObject = new MyGlobalObjectSem();
  replicationObject.setNetworkingObject(networkingObject);
  networkingObject.setReplicationObject(replicationObject);
  address = replicationObject.created();
}
```

Figure 15 The constructor of a global object

Then the methods of the semantic object should be exposed by the control object so that the application can access them. This could also be done very much by rule, which we can study in Figure 16. To begin with the method in the control object should generally return the same type, have the same name and the same parameters (type, order) as the method in the semantic object. It must also throw the same checked exceptions as the target method and in addition to them also `RemoteException`.

In the implementation of the method there can be three different cases that the replication object controls. The first two cases are only be relevant for invocations one a semantic object located on the same node while the third case is independent of the location of the semantic object. Still, the second case would allow the replication object to fetch a semantic object before proceeding with the local call. Despite the generality of the third case, it should be avoided if possible since it is heavy even for local calls.

The first case can for instance be used by the remote procedure call policy on the node with the semantic object. Since it doesn't want to add synchronization nor needs to track changes or running invocations in the semantic object threads can be allowed to proceed without involving the replication object at all. This case is probably not suitable for a policy that moves the semantic object since the policy cannot know if there are any ongoing invocations at any time and thus perhaps not know when it would be safe to move the semantic object.

The second case addresses this problem by allowing the replication object to count the ongoing invocations in the semantic object. At the same time the replication object can delay invocations while waiting for a current semantic object to let the invocation proceed on.

```
public void add(int index, Object element)
throws RemoteException {
  if (doLocal) { // This is strictly an optimization
    semanticObject.add(index, element);
    return;
  } else if (doFast && replicationObject.start(true)) {
    try {
      semanticObject.add(index, element);
      return;
    } finally {
      replicationObject.stop(true);
    }
  }
  InvocationResult result = replicationObject.invoke(
    true,
    "add(ILjava.lang.Object;)V",
    new Object[] {new Integer(index), element});
  if (result.isException()) {
    Object t = result.objectValue();
    if (t instanceof RuntimeException) {
      throw (RuntimeException) t;
    } else if (t instanceof Error) {
      throw (Error) t;
    } else {
      throw new UndeclaredThrowableException((Throwable) t);
    }
  }
  result.voidValue();
}
```

Figure 16 A method in a control object.

Obviously it necessary for the replication object to know when an invocation has terminated or the count will get wrong. This is protected in a `finally` clause since an exception is a normal way for an invocation to terminate that the application would expect to be able to recover from.

The second case is however insufficient for starting a remote procedure call, so also the third case is needed. In this case the replication object gets access to a String identifying the method and all arguments. The string and arguments should be serializable since the replication object might want to pass them to another node and invoke the method there, like a remote procedure call.

It is the control object's responsibility to ensure that the string and arguments passed to the replication object's `invoke` method can be recognized by its own `invoke` method. A part of the `invoke` method in the same class as the method in Figure 16 could look like the code in Figure 17. It first recognizes the string and the parameter array. Then the method is invoked on the semantic object. Finally an `InvocationResult` object describing the return value of the invocation is returned. The result could obviously also be an exception so exceptions have to be caught and packeted so that they can be passed back, perhaps to another node, and rethrown there.

```
...
if ("add(ILjava.lang.Object;)V".equals(name)
    && args.length == 2) {
  try {
    semanticObject.add(((Integer) args[0]).intValue(),
                       args[1]);
    return InvocationResult.voidResult();
  } catch (Exception e) {
    return InvocationResult.exceptionResult(e);
  } catch (PJOError pe) {
    return InvocationResult.exceptionResult(pe);
  } catch (Throwable t) {
    return InvocationResult.exceptionResult(
                                new PJOInvocationError(t));
  }
}
...
```

Figure 17 A part of the invoke method in a control object.

We have previously stated that the local representative of a control object must be unique on a node. The `Serializable` interface defines two methods that are useful to this end, `writeObject` and `readResolve`, which we strongly recommend that all control objects implement. These could, for example, follow the example in Figure 18.

The `GlobalObject` interface, which all control objects must implement, defines some more methods in addition to `invoke` but we believe it is obvious how they should be implemented and no special concerns about them to discuss.

```
private void writeObject(ObjectOutputStream out)
        throws IOException {
  if (address == null) {
    address = networkingObject.getNetworkAddress();
  }
  out.defaultWriteObject();
}
private Object readResolve() throws ObjectStreamException {
  ObjectManager manager = ObjectManager.getManager();
  GlobalObject obj;
  try {
    obj = manager.resolve(address, null, false);
  } catch (RemoteException re) {
    throw new InvalidObjectException("RemoteException: "
                                     + re);
  }
  if (obj == null) {
    return this;
  }
  if (obj.getClass() != getClass()) {
    throw new InvalidClassException("The global object of "
                + networkingObject.getNetworkAddress()
                + " is inconsistent with the preexisting "
                + obj);
  }
  return obj;
}
```
Figure 18 Example writeObject and readResolve methods.

3.7 Security

We chose the ostrich approach to security: put your head in the sand and pretend like it is raining. This approach might seem simplistic, but it has been shown that preventing a malicious adversary from being able to remove data from the network is expensive [56], and e.g. using strong encryption to guarantee privacy or authenticity is also expensive.

Thus we completely left those things out and assumed that the network could be trusted. It is then up to the user to punish anyone abusing the system, which we are well aware would be possible. Obviously anyone is welcome to device and add measures to restrict that trust.

3.8 Summary

In this section we have first discussed our choice of programming environment and a very high-level view on our chosen solution strategy. Then we discussed our object framework from a largely functional stand-point leading up to a description of the central parts of its API. By then we had completed the necessary groundwork and switched to discussing our collection framework, again working our way forward to the API.

The second half of this section we then spent on discussing more specific design issues. First out were issues surrounding the interaction with the overlay network, i.e. Tapestry. Then followed a discussion of the interactions within the object framework. Finally we had a close look on control objects and what needs to be known about them.

4 Implementation

We implemented a part of the system outlined in the previous section. It spanned all three layers, from dealing with an overlay network, via a rudimentary distributed object system to distributed collections. This implementation will be discussed more in detail in this section.

First a brief remainder about how things connect together, see Figure 19. Tapestry, the overlay network, is at the bottom and connects the network nodes, and on top of it is our thin adapter. Together these constitute the Overlay Network Layer that the Object Framework is built on.

In the Object Framework both the object manager and the networking objects communicate directly with the overlay network layer. The replication object manages the coherence of the object through configuring the control object to “do the right thing” and through collaborating with other replication objects, often replication objects part of another local representative of the same global object on a different node. The control objects are, in turn, both the facades of the global objects to the application and the holder of the semantic object, which implement the specific behaviour of the global object’s operations and contains its data.

The Collection Framework is not an entirely new level. Instead it contains a number of control object and semantic object classes that the application can instantiate, exactly like the Java Collections Framework.

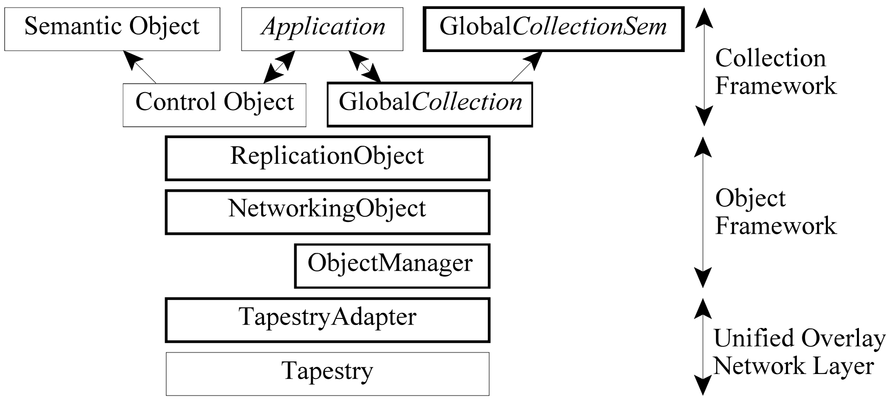

Figure 19 Detailed overview of the layers.

The numbers for our implementation are: It consists of 84 classes and interfaces and around 12 400 lines. Of these lines we use to estimate that roughly one third contain a statement, i.e. probably somewhere around 4 100 lines. Test code and programs used to measure the framework for Section 5 is not included in these numbers.

4.1 Network

On the lowest level of our implementation we used Tapestry. On top of this framework we added a thin layer exposing the functionality of it with an interface not entirely unlike the unified API for structured overlay networks [57].

The main reason for this organization was an uncertainty about which particular overlay we were going to use. Originally we looked at DKS and eventually this became Tapestry. This way the problem was contained and we could proceed with other work pending a final decision. It turned out that Tapestry didn’t provide a call similar to the range operation, and this way there was a natural place to add it. Another side effect is of course that currently there is a limited amount of code that would be affected by a change of the overlay. On the other hand it is reasonable to expect that it has a price in somewhat higher run-time overheads.

Tapestry itself is built on the seda/sandstorm framework [58]. The short story is that it subscribes very much to the “don’t call us, we’ll call you”-philosophy. We thought that it was pretty server-centric and that we more or less abused it since we weren’t prepared to fully force it on those using our framework. Its perhaps most special property is that it expects the application to work through passing messages between stages (essentially mini-servers) that perform some pretty short term computation on them. For starting the application the framework has provided a generic entry-point to which you would usually pass your XML like configuration file from which it sets up the stages, event queues, thread managers and so forth. We took some measures to hide some issues associated with that, e.g. created a generic stage that handles some initialization and can allow an application programmer to get started on “business code” rather than formalisms quicker. We don’t claim to have reached far enough to make it a simple environment that won’t haunt the careless programmer.

Another note about Tapestry is that it doesn’t use plain Java serialization to send messages. Instead it uses an explicit mechanism where objects have to serialize themselves onto a stream using operations on Java primitives, and subsequently deserialize themselves the same way. We guessed that this was a remainder from an earlier time when standard Java serialization was considered slow, but we didn’t want to commit to it now since serialization is nowadays considered fast. To avoid that we used a scheme that actually serializes the message twice, first using standard serialization into a byte stream and then writes out the byte stream. We don’t think this is appropriate for an actual implementation, there we believe Tapestry should either use standard serialization, one should take the hit of having to explicitly serialize one’s objects throughout or possibly consider using some other overlay network.

We would also like to discuss the thin layer between Tapestry and the object framework some more. To begin with we note that we used the opportunity to simplify the networking API. As just discussed we removed the requirement that messages must serialize themselves. We also hid some complexities surrounding the actual sending and reception of messages, e.g. some data structures that needed to be configured and traversed. We also decided to use a standard up-call mechanism to notify the application about events such as a received message or a change of the local ranges. A major part of our layer concerns the `range` operation which reveals the identifier ranges this node is responsible for. We needed it and had to implement it through direct examination of the routing table. In the already mentioned up-call we XORed the new ranges with the old ranges to obtain only those identifier ranges added and lost.

4.2 Object Framework

The part of the object framework that we implemented was functionally quite similar to Sun’s Java Remote Method Invocation (RMI) [29]. On a slightly abstract level it also works in much the same way as RMI even though there are differences. Specifically both frameworks have the notion of a stub object that is distributed to other nodes wishing to use an application/ semantic object and support access to remote/global objects through remote procedure calls. Thus both frameworks also support marshalling/unmarshalling of parameters and return values/exceptions, and both has the limitation that objects other than global objects are passed by value rather than by reference (we do however not strictly enforce this so changes to an object parameter might in some cases be visible to the caller).

We've already explained that our design and implementation to some extent followed ideas borrowed from the Globe project [3]. After this work we now feel that it is unclear if the networking object actually served any purpose in our implementation after all, given that also the local object manager is coupled with the networking layer to demultiplex incoming replication messages. Thus dropping a new networking object in place and expecting things to work might not be all there is to it for us.

We only implemented a replication object for a policy using only remote procedure calls. In this case, compared to the case for networking objects, it is much more clear that other replication objects could be developed and dropped in place and implement other policies, e.g. involving caching and replication of the semantic object, without greater side-effects. We had hoped to investigate those issues more, but that would have extended the scope of this work too much.

While designing and implementing the object framework we wanted to stay close to the Java object model. As discussed in Section 2.1, we think there are inherent differences between local objects and global objects that should not and cannot be hidden, and so there are differences.

One of these are the `equals` and `hashCode` operations required from every object. In the distributed case they are more difficult, in fact they cannot be used directly in RMI. The direct technical problem is that their formal specification doesn't allow them to throw some `RemoteException`. More serious are their semantic problems, in particular `equals`' but given their relation that also affects `hashCode`. Consider that the application should only interact with control objects, not the semantic objects, so it is the control object's `equals` function that is invoked. It might very well be invoked with another control object as argument, which the semantic object might know nothing about. If it doesn't then the control object would have broken the semantic object. The control object on the other hand cannot in general turn the parameter into some other object, such as the semantic object of another global object, since that object might very well be non-serializable and in some other JVM. Thus we concluded that we'd let control objects throw `UnsupportedOperationException` for both `equals` and `hashCode` if they wanted to.

What is similar between our framework, with the remote procedure call policy, and Java in general is the coherence model on objects. Java provides an at least release consistent memory model [59]. Our framework does not ensure that multiple accesses to an object are serialized and thus the programmer needs to ensure that there is sufficient synchronization within the program. Our framework does however issue both a local release and a local acquire before a remote method is invoked, so a remote node that knows one call has returned before another is invoked is assured that the new call will se a coherent view of local memory regardless of which service thread handles the latter invocation. Still, one need to consider both local calls and any possibility of recursion when one chose to rely on this over using synchronization primitives. Recursion between objects might in our framework need to be considered more like concurrent execution than is usually the case (see Section 3.1.4 and Figure 7).

A difference that must be kept in mind while working with concurrent execution in our framework is that a logical thread of control is not exactly the same thing as in standard single-JVM Java. There, a thread can always reacquire a lock it is already holding. In our framework calling a global object conceptually continues the same thread of execution, even if it is invoked in another JVM. The JVM doesn't quite see it this way though, and thus the thread might not be able to reacquire the lock if the other global object calls back recursively. Dead-lock might very well follow, but can be avoided with some planning. This limitation is also shared with RMI.

Another spin on this single logical thread view are exceptions from invocations on global objects. As discussed in Section 3.1.3 the call-stack of an exception can be rewritten to bind together the procedure calls and hide framework administration in between. We noted that Sun opened for implementing it in their Java standard library and we implemented it in a limited fashion. One obvious limitation is that we don't rewrite the call stack of contained (chained) exceptions. We found it disturbing that Sun didn't open for adding new information to the stack trace, such as the node a particular element in the trace was executed on or perhaps a pseudo-element to represent a node jump. Location might be an important factor so we have mixed feelings about rewriting.

4.3 Collection framework

We started out from the interfaces Sun made for the Java Collections Framework (JCF) [1] in our work with collections. We then made some changes to these to suit our environment, e.g. `equals` and `hashCode` were unrequired, for the reasons in Section 4.2, and we added `RemoteException` to the exception specification of all operations. Our interface structure, see Figure 20, was thus in part very similar to JCF.

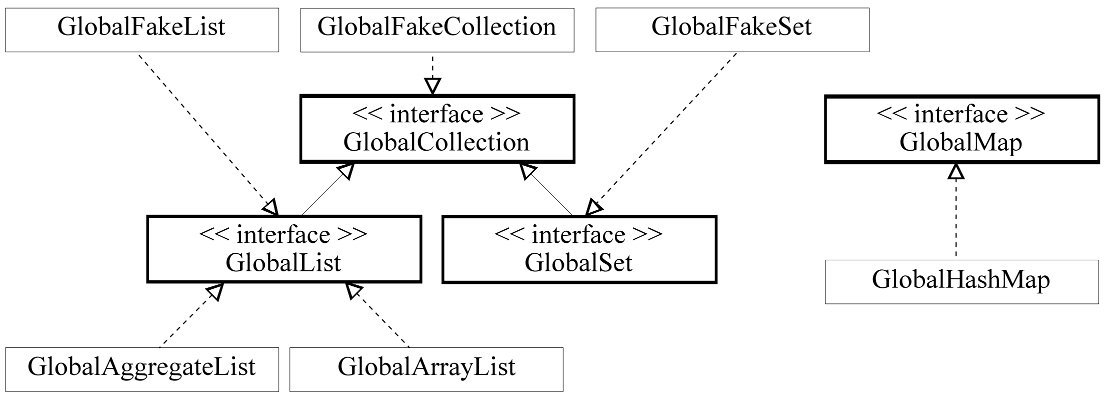
GlobalFakeList
GlobalFakeCollection
GlobalFakeSet
<< interface >> GlobalCollection
<< interface >> GlobalMap
<< interface >> GlobalList
<< interface >> GlobalSet
GlobalHashMap
GlobalAggregateList
GlobalArrayList

Figure 20 Classes and interfaces in our collection framework.

Then we created some wrappers, `GlobalArrayList` and `GlobalHashMap`, around actual classes from the JCF to have a basic set of collections to work with. These are monolithic collections, so they were not sufficient to satisfy our goal of collections where fragments of the collections are located on different nodes. A further problem is the habit in JCF to return entangled objects, e.g. iterators, sub-lists, etc, which breaks if the collection object and, let's assume, the iterator would reside in different JVMs. It is even difficult to move entangled objects between JVMs and preserve the entanglement.

That was a problem that we didn't really have to solve since the replication policy we used, only remote procedure calls, guaranteed that the collection and all sub-objects (`GlobalFakeList`, `GlobalFakeCollection`, `GlobalFakeSet`, `GlobalFakeIterator`, etc[4]) would forever remain within the same JVM. Thus requiring that the node never leaves the network, or at least not until the run is over.

Building on the wrapped array list, we constructed an aggregate list class. It contains a list of sub-lists and provides a linear view on them, i.e. something that appears like the concatenation of them. This list has the special `append` operation to add a new sub-list to the end of it.

We had in particular two reasons not to directly add a list to the internal list of sub-lists. The primary motivation was that in our view JCF wouldn't behave that way, essentially assuming ownership of the collection. A secondary concern was the safety issues surrounding some caching that the aggregate list had to perform for practical reasons. The JCF has a very index centric API and thus many operations depend on that the sub-list responsible for a particular index can be found fast. We thus felt that the list had to cache the sizes of the sub-lists, and thus it is imperative that the sub-lists are not accessed except through the list.

Following the JCF our collections must be externally synchronized whenever there might be a race involving a structurally modifying operation. We didn't provide any means for that, but it should not be too difficult with moderate requirements on safety, scalability and performance.

Alternatives that could be considered in future work is of course to swallow the list given to the `append` operation and inform the user that he must not use that list anymore after that. Then he could choose to pass a copy and the pay price of that if necessary, or give up the list and pay a lower price. Another design that might be worth exploring is the workings of iterators. Collections should perhaps have additional operations in their interface, either on interface or implementation level, and the iterator object act more like a proxy object into the collection but on the object framework level rather than behind the scenes in the JVM. For distributed collections one might also want iterators that act like floating cursors in the collection rather than being invalidated whenever the collection is modified, also something that would probably be easier if the iterator in some sense were internal to the collection.

4.4 Summary

In this section we have discussed implementation issues that we faced during the implementation of our prototype. We have discussed issues with our middleware on top of Tapestry, the seda framework, serialization and up-calls (the observer pattern). We have also discussed the object framework, the problems with `equals` and `hashCode`, related the memory model of the remote procedure call policy to that of Java, and discussed threads and exceptions. Finally we discussed the collection framework, wrapped collections, semantic object entanglement and the `append` operation on `GlobalAggregateList`.

[4] The "fake" in their names refers only to that their semantic objects are entangled objects and thus might depend on the replication policy also for correctness.

5 Evaluation

In this section we will show some evaluation results on this framework. First we briefly investigate some fundamental properties of the overlay network and our object framework. They set some important boundaries for what is possible to achieve in the collection framework since the collection framework is built on top of them. Then we proceed by analysing the behaviour of `GlobalAggregateList`, which is the actual class that has most of the properties we've been designing. Finally we consider a program sorting `String` objects, as an example of something like actual work being performed by the framework.

5.1 Test platform

We have measured the performance of some operations in our framework on up to 8 Sun SunBlade 100 Sparc computers running Solaris 5.9 connected with a 100 Mbps ethernet network through a router. The measurements were not run in a clean environment but there were other unrelated programs running in parallel on the machines and unrelated traffic on the network. We believe that these measurements still have some validity both since the measurements were performed at a time when the amount of related activity was expected to be low and since peer-to-peer frameworks often target such dirty environments, e.g. running in the background with low priority while the machine is primarily used for other activities.

5.2 Test method

We measured our framework by creating small test programs and created instances of Sun's `java.util.Date` class at some points to obtain time stamps from when execution passed those points. In the particular environment used these provides measurements with millisecond resolution. It does not compensate for instance for the operating system momentarily interrupting and working on another task.

We estimated the mean overhead of this method to 0.004 ms with a standard deviation of 0.077 ms by looping over the creation of two `Date` objects and checking the difference between them.

5.3 Time by architectural layer

To begin with we wanted to break down the time spent in different layers of our implementation. Thus, we estimated the actual network latency by using the standard ping utility on the test platform. Then we created a ping program directly on top of Tapestry to minimize the additional overhead when measuring the round-trip time. Finally we created a minimal global object and measured the time to invoke and return from a method that did nothing, on a remote object using the remote procedure call policy on top of Tapestry.

The results of the measurements are shown in Table 1. In the ping measurement the first sample was ignored since it is known to be non-representative, which is marked by the asterisk (*) after the name. We assumed that the round-trip times were approximately normally distributed around the mean value and computed the variance of our data values. Then we plotted the mean round-trip time with the three standard deviations (3σ) interval in Figure 21 (the time must obviously be non-negative). We note that if the assumption would have been accurate then there would only be a 0.13 % probability to obtain a value outside the mean$\pm 3\sigma$ interval. For the system ping utility it is not clear that we have seen improbable values because of the limited clock resolution, but in both the other cases we got values far outside the interval thus contradicting the assumption.

It is evident from this test that both Tapestry and our object framework add a considerable overhead on top of the network latency. The ping value is however lower than could possibly be achieved by any application, since it can be handled entirely by the operating system on the remote node.

	mean	median	min	max	stddev	n
System Ping*	0.04	0	0	1	0.20	99
Tapestry Ping	16.45	12	9	131	14.07	100
void func()	27.45	25	16	147	15.23	100

Table 1 Round-trip statistics (ms)

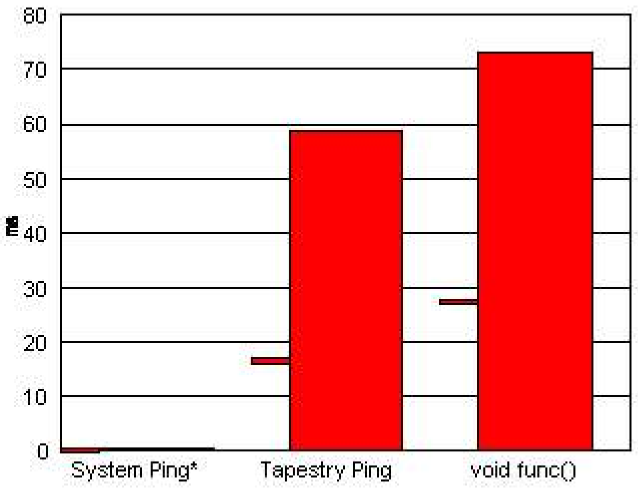

Figure 21 Mean round-trip time and *3*σ interval

5.4 GlobalAggregateList

We felt that `GlobalAggregateList` was the most interesting of our collection classes to measure since it was the only one that could form collections spanning multiple nodes.

5.4.1 Maximal collection size

Our first test was to construct and measure how large collections could be created. We did this by creating one `GlobalArrayList` on each node, adding some number of *different* `String` instances (all of length 13) and appending them to the `GlobalArrayList`. The largest number of strings that could be added without suffering an `OutOfMemoryError` (which in turn tended to be seen as `IllegalArgumentException` exceptions from the `append` call).

We measured the result for 1 through 8 node configurations. For comparison we also measured the largest `ArrayList` that could be created in a JVM with the same memory restrictions but without any of our framework nor with Tapestry loaded. The result is shown in Table 2 and further illustrated in Figure 22. As expected the maximum size of the `GlobalAggregateList` grows almost linearly with the number of nodes and the maximum size of the `ArrayList` is constant as it cannot take advantage of more nodes. The decreasing size of the `GlobalArrayList` we believe is largely due to increasing overheads in Tapestry when the number of connected nodes increase.

Tapestry and our framework incur a memory overhead that limits the maximal collection size, as could be expected. This is easily seen in the single node case where the maximal collection size is higher for `ArrayList` than for `GlobalAggregateList`. We anticipate that other maximal sizes could be found if adding elements to the `GlobalAggregateList` using other methods, e.g. `add`.

Obviously the exact value of the maximum size is not very interesting on its own. Rather we think that the proportions between the maximum sizes are more likely to generalize to other object sizes. Our motivation for this is that if the collection contains 980 000 elements then memory is consumed by 980 000 Strings in addition to the collection and other overhead. Still, if one has a collection one probably want to put something in it so measuring maximal size with a single element contained many times (e.g. null) would perhaps be even more misleading.

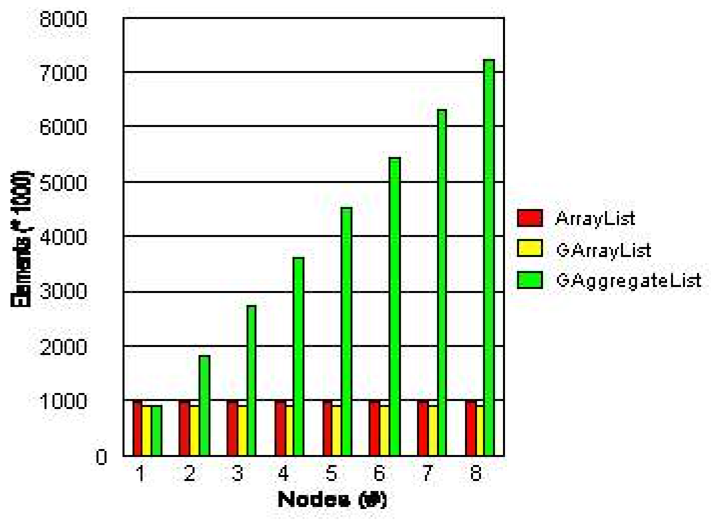

Figure 22 Maximal collection sizes

Nodes	Array List	Global Array List	Global Aggregate List
1	980000	906000	906000
2	980000	906000	1812000
3	980000	905000	2715000
4	980000	905000	3620000
5	980000	904000	4520000
6	980000	904000	5424000
7	980000	904000	6328000
8	980000	903000	7224000

Table 2 Maximal collection sizes

5.4.2 Time to create a collection

Talking about the time to construct such a global collection `GlobalAggregateList` is a bit difficult. We lacked the tools to measure actual executions with any precision, primarily a synchronization mechanism around the `append` operation. We can however offer some speculation on the matter.

We based our speculations on strictly sequential measures. So we instrumented the program, previously used to measure the maximal collection sizes, to time the creation and the append with a controlled size of the list that it created and appended, e.g. if two nodes append a list with 490 000 elements then the total size will become 980 000 elements. It was sequential because we first started the first node and let it complete, then the next node, and so on.

Then we constructed speculative parallel traces, like the one shown in Figure 23, using the obtained task times for different number of nodes and collection sizes. We counted the leading add phase as fully parallelizable since it was performed on a co-located `GlobalArrayList` on each node (thus different lists). The succeeding append-phase should be regarded as sequential, since the design of `append` would require the node to obtain some kind of exclusive lock before invoking `append` and hold it for the duration of the call.

We do not intend to provide an answer to how the start of the collection creation should be synchronized. We also note that this kind of speculation disregards some additional overhead that a properly synchronized program would have, although we believe that it could to some extent be done in parallel and thus not affect the total time quite as much as it might at first appear.

The perhaps most obvious conclusion that can be drawn from the diagram in Figure 23 is that `append` is a problem. One should not be deceived into believing that it accounts for an as great proportion of the time as in that diagram, intuitively the proportion grows with the number of nodes both since the add-phase becomes shorter and since there are more lists to append. In Section 4.3 we suggested that one instead swallow the list given as parameter, which would greatly reduce the cost of the operation, down to around the cost of a two simple remote function calls. One could also consider leaving JCF's unsynchronized principle and make some part of append synchronized while still allowing the list cloning to be parallel. Then it might be much more difficult to speculate a trace like this, but we believe that it might increase parallelism and reduce the total time considerably although perhaps less than our former suggestion.

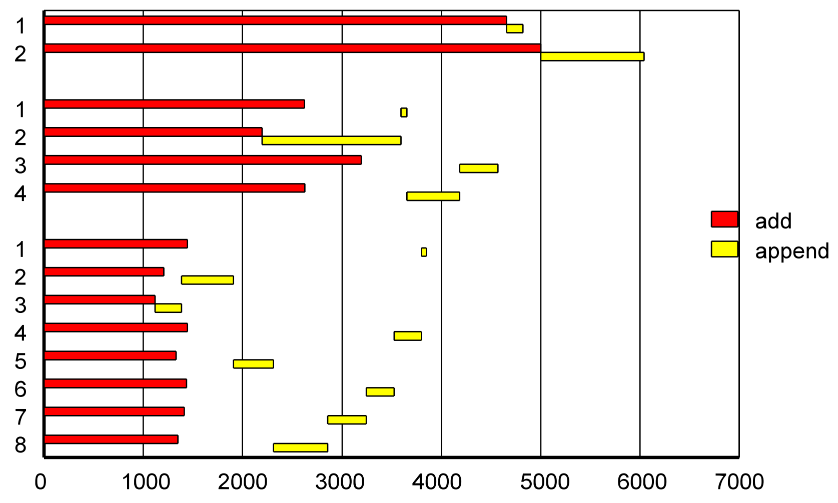

Figure 23 Speculated traces of collection creation in 2-, 4- and 8-node configurations.

Still, constructing such traces from a number of executions, we came up with the times shown in Figure 24. The values for N=1 were however measured on an `ArrayList` which we chose to consider "the best sequential program" here, and are thus real times. Average construction times per collection size and number of nodes are listed in Table 7 in Appendix D. The figure shows individual outcomes to also give some idea of the variance. Note also that the garbage collector is a source of non-normal distributed noise.

Here we only considered the creation of a collection but so far our collection appears competitive, at least for very large collections. Based on these numbers and speculation we also think that there is a great potential for lowering the break-even point with `ArrayList` in the future.

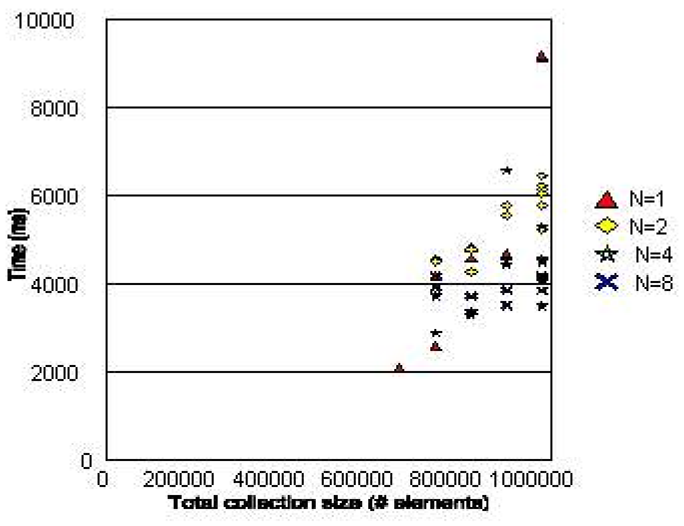

Figure 24 Speculated total time to construct collections

5.4.3 Time of collection operations

The entire point in collections might however not be their pure existence. They must necessarily exist if one is to move on to using them, but why bother unless one can? So let's look at a few operations that `GlobalAggregateList` supports. We chose to look at the `contains`, `get`, and `indexOf` operations which we felt was a reasonable trade-off between the number of operations that the results would apply to and the amount of work required.

We measured these operations by again creating a `GlobalAggregateList`, evenly distributed over the nodes. Then we timed blocks of 100 calls to each method with random parameters from the set of values for which the function succeeds (i.e. `String` instances in the list and valid indices). Each `String` was unique within the list, and 13 characters long.

We have plotted the average time used by `contains` as a function of number of nodes and collection size in Figure 25, also listed in Table 3. It is apparent that the step to start using actually remote procedures is expensive and with only a single extra processing unit it is hard to make up for that.

This figure does not conclusively show that adding more and more nodes will at some point start to increase the time-consumption. This might be what happens for the smallest collection in the 8-node configuration, and we would certainly expect to find this effect if sufficiently many nodes were used. There are several reasons for this, the most notable one being that when the number of sub-lists exceeds the maximum number of simultaneous requests the system can handle then requests will have to wait for previous requests to finish before they can be issued. Another reason might be that the routing time in the network increases with the number of nodes and the actual execution time of the operation eventually becomes a so small portion of the total invocation time that even if it is reduced the total time still increases.

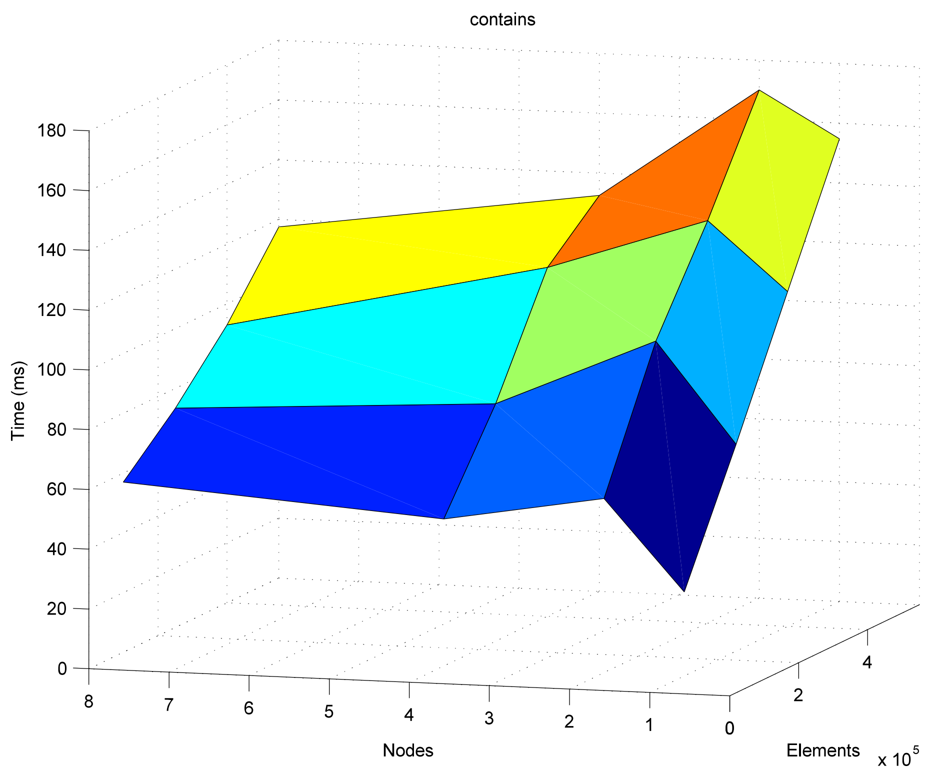

Figure 25 Average time of contains by number of nodes and size of collection

Other functions which we expect to behave similarly to contains include `clear`, `removeAll` and `retainAll`. The exploited characteristic of these operations is that they can be invoked in parallel on the sub-lists. `containsAll` does not have that property since the collection might be partially contained by several sub-lists, only together containing the entire collection.

`indexOf` behaves in a distinctly different way from `contains`, shown in Figure 26 and listed in Table 4. This is not inherent from the operation as such, which could be implemented using a technique similar to that of `contains`. That would however not be possible for functions that must traverse the list in order from the beginning to the end, e.g. `remove(Object)`implemented in terms of `remove(Object)` calls on the sub-lists. Such functions should get essentially this graph. Obviously we would also expect `lastIndexOf` to behave similarly to `indexOf`, whichever way they are implemented.

Table 3 Average time (ms) of GlobalAggregateList's contains operation.

		Collection size			
		100000	250000	400000	550000
	1	28,0	69,1	112,0	154,7
	2	58,1	102,5	134,6	170,0
	4	49,1	79,3	116,7	132,4
Nodes	8	56,9	73,4	92,8	117,4

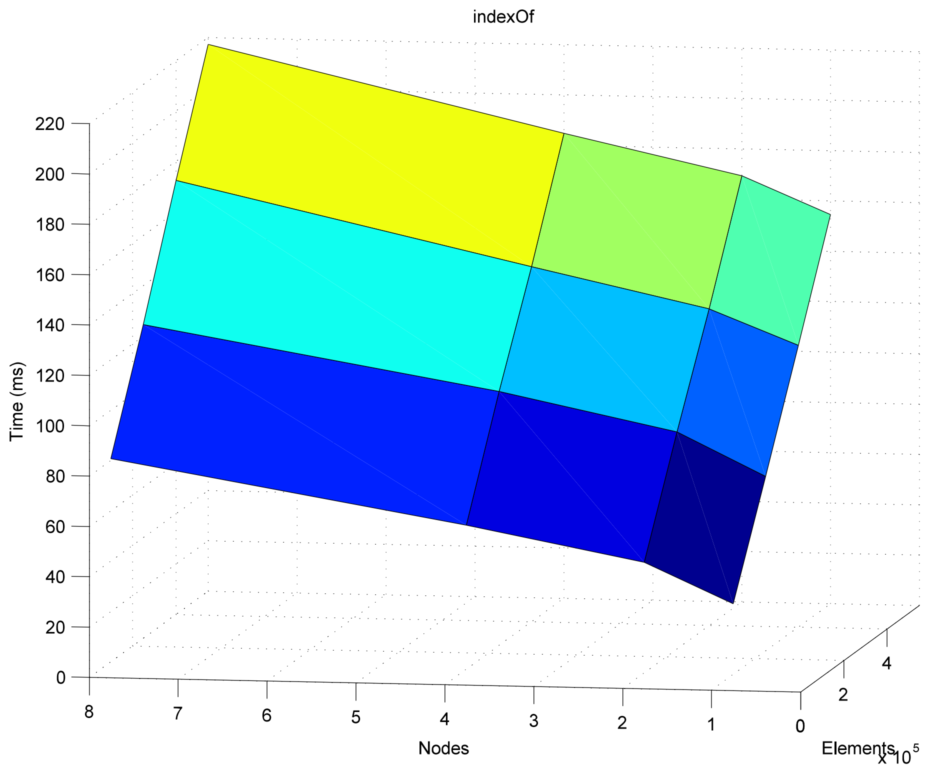

Figure 26 Average time of indexOf by number of nodes and size of collection

Finally we had the `get` operation, which has yet another distinctly different behaviour shown in Figure 27 and listed in Table 5. It is quite independent of the size of the collection, which we expected. We believe that the addition work that makes the time increase as the number of nodes increases beyond two is the search for the right sub-list in `GlobalAggregateList`, we used a linear search which could be improved for instance by using binary search.

We further believe that the `get` operation's behaviour is similar to the behaviour of `add(Object)`, `add(int, Object)`, `addAll(Collection)`, `addAll(int, Collection)`, `remove(int)` and `set(int, Object)`. `add(Object)` and `addAll(Collection)` might be slightly different since they don't have to search for the sub-list, so possibly they will even be approximately constant in the number of nodes. The other operations share the characteristic that the sub-list responsible for a given index has to be found and an operation invoked recursively.

Table 4 Average time (ms) of GlobalAggregateList's indexOf operation.

		Collection size			
		100000	250000	400000	550000
Nodes	1	28,1	69,2	111,9	154,5
	2	43,7	86,2	125,8	169,2
	4	57,2	100,8	141,0	184,7
	8	80,7	124,6	172,4	217,1

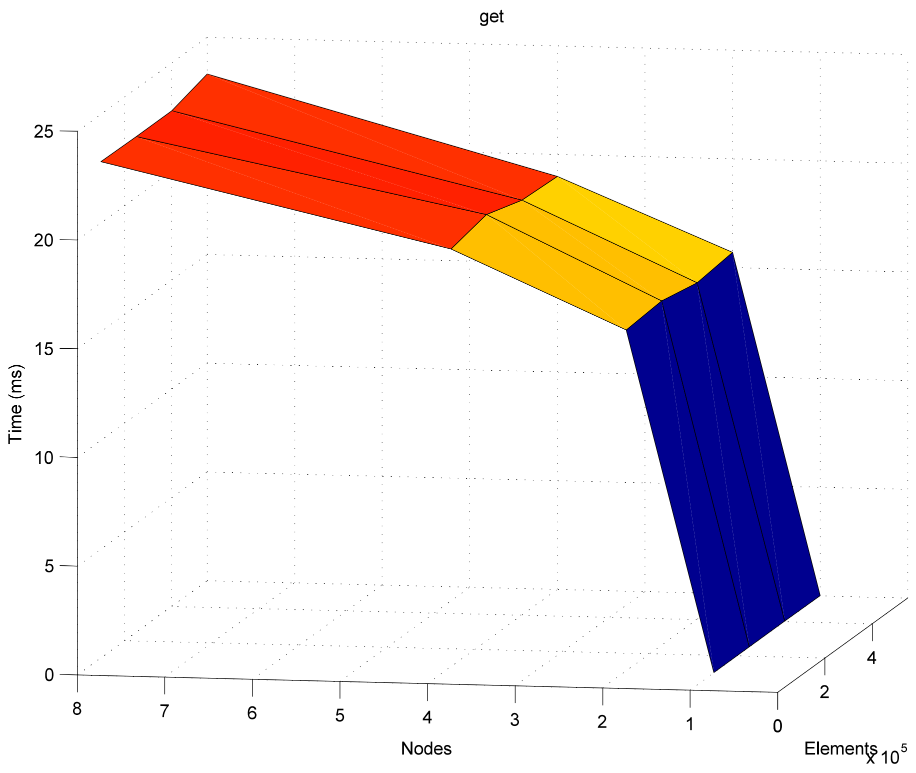

Figure 27 Average time of get by number of nodes and collection size

Table 5 Average time (ms) of GlobalAggregateList's get operation.

		Collection size			
		100000	250000	400000	550000
	1	0,019	0,023	0,015	0,017
	2	15,7	15,8	15,5	15,7
	4	19,2	19,6	19,1	19,0
Nodes	8	22,8	22,8	22,8	23,3

5.5 Example program

We created an utterly simple demonstration of the framework. It creates a `GlobalAggregateList` containing 512 `String` objects chosen fairly randomly from the set of strings that can be created from `byte` arrays with 13 coordinates. This list is evenly distributed over the chosen number of nodes. Then the list is sorted using a quick-sort algorithm, for simplicity, and finally the sort is verified to be correct.

A particular note about the algorithm, see Figure 28, is that it does not directly access the list during the sort. Instead it first collects all elements to a local collection using a single list operation (`toArray`) and when the local array is sorted it uses two other bulk operations to put the sorted elements back into the list in order (`clear`/`addAll`, the Update phase). This is possible while the list is small and makes a huge difference for time consumption. Using such a simple sequential algorithm there was no hope that it could possibly compare to a local solution in terms of time-consumption, but the point was just to show that it could sort.

```
// The Gather - sort phase
Object strings[] = gaList.toArray();
for (int i = 0; i < m; i++) {
  String s0 = (String) strings[i];
  String r = s0;
  int min = I;
  for (int j = I + 1; j < m; j++) {
    String s1 = (String) strings[j];
    if (s0.compareTo(s1) > 0) {
      min = j;
      s0 = s1;
    }
  }
  if (min != I) {
    strings[min] = r;
    strings[i] = s0;
  }
}
```

```
// The update phase
gaList.clear();
gaList.addAll(Arrays.asList(strings));
```

Figure 28 The sort program

Although time was not our main objective here, we still think it is interesting to look briefly at the time used by this program. We show the results from a series of test runs in Figure 29 and list average times in Table 6. It is evident that it is an advantage to have fewer nodes in this program, since then it is easier to collect the data and there is no distributed computation to make up for that. It is not surprising that there is a big difference between one and two nodes.

The gather - sort time in the one node case provides an upper bound on the time to sort the list and it is reasonable to believe that the additional time on more nodes is purely expended on gathering the data. It might seem surprising that the update time is constant, but we explain that with that the `addAll` operation used always adds the entire collection to a single sub-list of the `GlobalAggregateList` (which thus has a monolithic distribution after the sort). That the gather time still increases slower than the number of nodes we believe is due to parallelism in the algorithm, which can have up to 10 simultaneous pending requests. The gather time might increase faster with the number of nodes once that threshold is reached, or at some point thereafter.

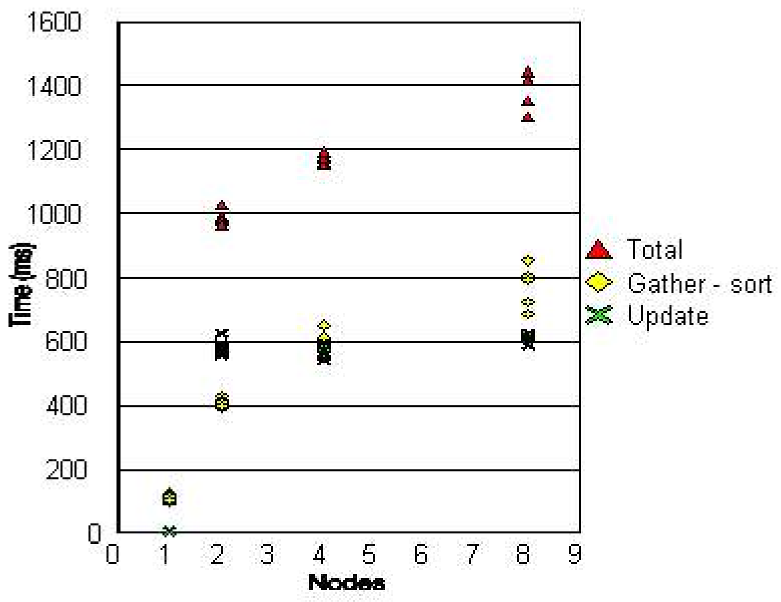

Figure 29 Times to sort 512 Strings

Table 6 Average times (ms) to sort 512 Strings.

Nodes	Gather-sort	Update	Total
1	105,8	6,1	111,9
2	407,4	575,6	983,1
4	604,3	572,8	1177,1
8	774,4	612,2	1386,6

5.6 Summary

In this section we have evaluated our framework, with particular focus on `GlobalAggregateList`, from a performance stand-point. First we measured round-trip times over the network, Tapestry and our object framework to have some sense of how much time is spent in lower layers. From this we deduced that the overhead of a remote procedure call was approximately 25 ms in the test environment. We did attribute 12 ms to Tapestry and other lower levels that were not part of our undertaking, thus leaving 52 % of the time that one could try to reduce.

Then we looked at the maximal storage capacity of `GlobalAggregateList` and related it to `java.util.ArrayList`. This showed that our framework and Tapestry have some memory overhead reducing the maximal capacity on a single node. It also showed that the maximum size of the `GlobalAggregateList` grew linearly with the number of nodes in the tested configurations. With only two nodes it exceeded the `ArrayList` list’s maximum capacity by 84 %.

We have speculated on the time required to construct `GlobalArrayList` collections, of different size and over different numbers of nodes, by constructing possible parallel traces based on measurements on sequential executions. In these traces, `append` accounted for 10 - 73 % of the time, increasing with the number of nodes.

We also examined the time characteristics of three operations in `GlobalArrayList` and found that these were distinctly different. They differed both in how the size of the collection and the number of participating nodes affected the invocation times. The number of nodes influenced the results in all cases, while one operation was insensitive to the size of the collection. We discussed which operations could be expected to have similar time characteristics to the examined operations, together covering most operations of `GlobalArrayList`.

Finally we looked at a small and naive centralized sorting program. The performance results were not great, between 1.0 and 1.4 seconds in the distributed cases to sort 512 short strings, but it was intended to show that it worked and it did.

6 Possible applications

One can imagine different potential applications for a real implementation of a framework such as the one we have investigated. Common characteristics that we looked for are that the application should be distributed and share some collection(s) of objects between the nodes. We did not care about whether if our prototype implementation would already support any particular application.

6.1 The bank manager

The bank-manager example often used to illustrate CORBA is an application that we think is pretty much ideal for our framework, with a little twist. The situation is that a bank wants to manage its accounts through terminals connected to some account back-end. The usual solution is to build a client-server solution with a central bank-manager and clients accessing the bank manager using CORBA, or RMI for that matter.

Lets now add the twist, suppose also that there are several bank offices in the bank and each want to store their own accounts, but they also want to be able to access any account from every office. We have illustrated a situation like this in Figure 30, where there are three bank offices A, B and C. In each office, there is one bank manager and two terminals for accessing the managers, although there could as well be offices without their own bank manager or with several as well as offices with any reasonable number of terminals.

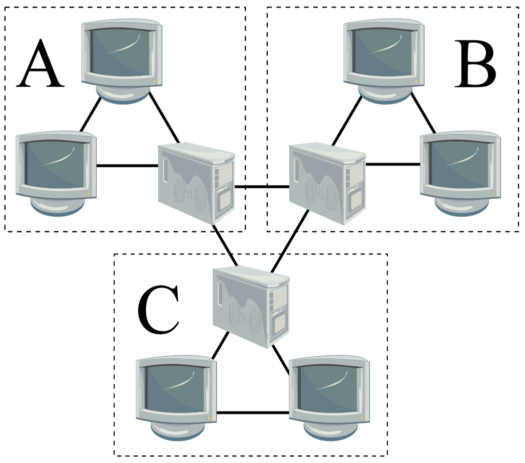

Figure 30 A distributed bank with three offices, each with a manager and terminals.

With a collection framework like ours, the offices could organize their accounts on their own bank manager into a single distributed collection. This way the accounts could be accessed in a uniform way from every terminal in the system, but each office still managing its own accounts could still feel confident in its control over them. This control would be complete if the accounts were themselves distributed objects using the remote procedure call replication policy in which case they would never actually leave the local bank manager.

In a more serious bank example there would be tough requirements on authentication and privacy protection. This is something that we don't provide any means for but it would, for instance, be possible to digitally sign the parameters and return values of function calls on the application level and use encrypted tunnels over an unprotected or weakly protected network. Certainly other options could also be designed and implemented without mixing this kind of functionality with core object or collection services.

6.2 Mail and messaging

Another setting where our framework could be used is an internet mail like application. With a minor modification described at the end this example could also be described in term of messaging applications.

Consider a distributed collection over a peer-to-peer network, such as the leftmost in Figure 31. Let there be an entry, say a mail-box, in that collection for each mail-address and that entry contain the e-mail address and a collection of mail folders, each of which consists of the name of the folder and a pointer to a collection containing the actual mails.

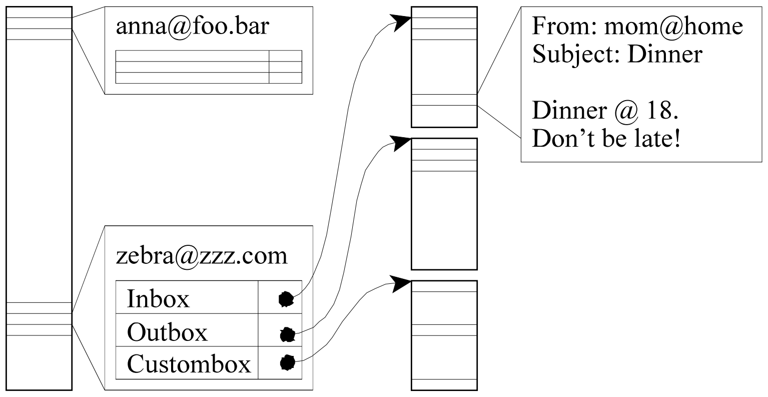

Figure 31 An e-mail like application.

Building an e-mail client that could locate your mail-box and read your mails would be quite trivial, at least from the perspective of network programming. Our framework would not help with any user interaction problem.

You would probably not want just anybody to reorganize your mail, so some protection should be used. One obvious possibility is to make your mail-box reside on a trusted node, thus only be accessible through RPC to prevent one possibility for introspection, and then only allow the adding of references to mails to the inbox without showing the mail-box an authentication token. One can consider if the mail pointer should include more information, e.g. to reduce spam problems but that is outside our scope.

Finally we will extend this example slightly to cope with messaging too. Add a new collection to the mail-box, or let one of the mail-folders be a *not-a-mail-folder folder*. Say that your messaging client adds a global object with the remote procedure call replication policy resident on your computer and implementing some call-back interface to this collection. The mail-box could then allow this collection to be traversed and some call-back method called, which in turn leads to that the call-back method is executed on your computer and the message instantly delivered.

6.3 Collaborative drawing tool

The final application where we chose to discuss how our framework could be applied is a collaborative drawing tool, i.e. a tool providing a shared canvas that people in different location can be painting on concurrently. This application, as seen from the collection framework point of view, is remarkably similar to the mail/messaging application.

Analogously to before, we assumed that there was a distributed collection holding all drawings and that the drawing contained sufficient information for it be found. In this case we chose to give the drawing a name, some content, a list of clients observing it and any extra meta-data needed to find the drawing. This is the situation shown in Figure ?.

A drawing could potentially contain a wide variety of different figures, e.g. text, rectangles, ellipses, polygons, etc. There is some sample contents shown in the figure, although many other kinds of figures would be possible and each figure would contain additional information, such as position and size, z-order, the colour of its details perhaps, etc.

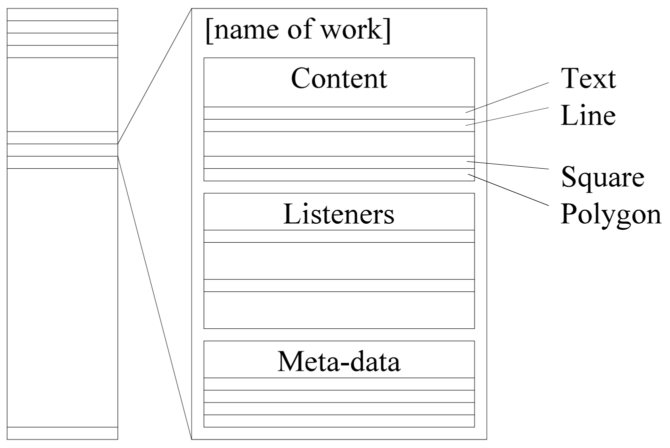

Figure 32 Structure of a drawing program.

Similarly to the messaging example, it would be nice if changes to a drawing would be reflected in all clients monitoring it. The same kind of listener structure would be applicable here too. A messaging client would typically be expected to have the single `message` operation but a drawing could be changed in at least three different ways, things could be added, things could be removed or things could get their attributes changed. Our framework would handle this equally gracefully whether one prefer sending event objects to a single call-back method or to different depending on event.

Another decision that would have to be made is whether if figure objects should be global objects or just ordinary serializable objects. We believe that without more advanced replication policies the latter might be preferable combined with some application level caching and clever coding of the figures' identities and z-order. We would hope that more advanced replication policies be deviced that could alleviate the application programmer from these concerns.

6.4 Summary

We have briefly discussed how our framework could be used in a few different applications. The first were an extended version of the bank manager commonly used to illustrate CORBA, the second an e-mail and messaging application and the third a collaborative drawing tool. While the discussions have been brief we believe we have shown that our framework has potential for working in these applications.

7 Conclusions

In this project we have surveyed a wide range of approaches to software distributed shared memory ranging from linear page-based memory to object memory. We have also studied consistency models and consistency protocols that implement them. In addition to this we have studied several kinds of both local and distributed collection frameworks. A summary of these studies has been presented as related work in Section 2.

Based on the studies mentioned above and our analysis of existing approaches to data collections, we have designed a distributed collection framework with essentially the same API as the Java Collections Framework of the Java 2 Platform, Standard Edition (J2SE). The J2SE Collections Framework includes six collection interfaces (Collection, List, Set, SortedSet, Map and SortedMap) and several general-purpose implementations of the interfaces such as HashSet, LinkedList and HashMap.

The major goal and accomplishment in our framework is the ability to create collections partitioned over several collaborating nodes in a network, a.k.a. distributed collections. This is significantly different from for instance the Java Collections Framework where a collection is confined to a single node. We believe that the distributed collection framework API proposed in this thesis is so similar to the API of the Java Collections Framework that someone familiar with the latter could also program using our distributed framework. That is on the other hand different from tuple space or database programming which use different primitives with different semantics.

We have also designed a distributed object framework using peer-to-peer technology to support the collection framework. For the object framework we used ideas from Vrije Universiteit’s Globe project to enable it to support experiments with explicitly controlled coherence and consistency. We believe this has made our framework better suited for such experiments than for instance Java Remote Method Invocation, because it provides clear interfaces for developing new replication policies and allows existing infrastructure to be reused. Our collections were designed to be distributed objects in this framework, and the distributed collections designed as data structures made up of distributed objects.

In order to test and evaluate our approach, we have implemented a basic prototype of the distributed collection framework. Our evaluation shows that it is possible to use the API for a distributed collection framework, but also that there are points at which the API might be impractical and it might be worthwhile to further explore alternative solutions. Among these we particularly note the semantic object entanglement between collections and iterators (and other views on the collection), indeed even that the view is invalidated whenever the collection is structurally modified.

The evaluation also showed that our prototype implementation of a partitioned collection was significantly slower than a local collection, unless the collections were very large. This is obviously as could be expected as the collection itself executes the same code but with additional book-keeping and communication costs. This can be made up for if either the local node is highly loaded, e.g. by the garbage collector if it is low on memory, if operations can be parallelized enough or if the collection simply wouldn’t fit on a single node.

We have reason to believe that the overhead of the framework could be significantly reduced in future implementations, for instance by removing the double serialization of messages that we used because Tapestry didn’t support standard serialization or removing the networking object. This is part of the 52 % of the call overhead that is spent in our framework rather than in Tapestry or other parts of the communication.

In terms of scalability we found that our distributed collection could contain a number of elements linearly increasing with the number of nodes in the network. Still, the design of the append operation that we used in the prototype to construct distributed collections was identified as a scalability risk and could consume up to 73 % of the construction time in an 8-node configuration, increasing rapidly with the number of nodes.

Finally, we believe that the proposed distributed collection framework may be of interest to programmers of distributed applications and developers of distributed middleware.

8 Future work

Future work after us can build on either of our two main tracks, the distributed object framework or the distributed collection framework. We see potential for both theoretical and practical advances in both areas, and since they build on each other advances on one might also benefit the other.

We were particularly interested in further enhancement of the object framework. We would like to see more replication objects designed and implemented, especially involving caching and replication of the semantic object on several network nodes and exposing an API for controlling the coherence of the object. Another area of replication objects that could be explored is compound replication objects that take properties of the function call, e.g. name and parameters, into consideration when choosing replication policy. On the other hand we believe that the networking object could, and thus should, be removed since we did not find any use of it.

The base object framework services will also need to be fully developed, for instance remains to solve how to ensure that the local representatives will be available on their new home node after a node leaves the network. We have also in most cases only implemented the parts of the protocols outlined in this report that we needed for our evaluation, for example it might be good to be able to delete objects.

On the distributed collection framework we believe that further studies evaluating some alternative design choices would be in order. We have discussed some observations on this in Sections 4 and 5 on our experiences from the implementation and evaluation of our prototype.

Finally we note that the prototype implemented by us is just that, a prototype and not an industry strength object and collection framework. It does not or only partially implement several parts of the design we describe, some of which would probably be required by anyone seeking to develop applications rather than experimenting.

References

[1] Sun Microsystems Inc. “The Collections Framework” http://java.sun.com/j2se/1.4.2/docs/guide/collections/

[2] Silicon Graphics Inc. “SIG - Services & Support: Standard Template Library Programmer's Guide” http://www.sgi.com/tech/stl/

[3] M. van Steen, P. Homburg, and A. S. Tanenbaum. “Globe: A Wide-Area Distributed System” *IEEE Concurrency* January-March, 1999, pp. 70-78

[4] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph and J. Kubiatowicz. “Tapestry: A Resilient Global-Scale Overlay for Service Deployment” *IEEE Journal for Selected Areas in Communications* 2004, Vol. 22, Issue 1, pp 41-53

[5] K. Li and P. Hudak. “Memory Coherence in Shared Virtual Memory Systems” *ACM trans. on Computer Systems* Vol. 7 Issue 4, pp 321-359, November 1989

[6] J. Waldo, G. Wyant, A. Wollrath and S. Kendall. “A Note on Distributed Computing” *Sun Microsystems Technical Reports*, November 1994.

[7] L. Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” *IEEE Transactions on Computers*, Vol. 28 Issue 9, pp 690-691, September 1979

[8] J. Goodman. “Cache Consistency and Sequential Consistency” *Technical Report 61 IEEE Scalable Coherent Interface Working Group*, 1989

[9] M. Dubois, C. Scheurich and F. Briggs. “Memory Access Buffering in Multiprocessors” *Proceedings of the 13th Annual Symposium on Computer Architecture* 1986, pp 434-442

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy. “Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors” *Proceedings of the 17th Annual Symposium on Computer Architecture* 1990, pp 15-26

[11] B. Bershad, M. Zekauskas and W. Sawdon. “The Midway Distributed Shared Memory System” *Carnegie Mellon University Technical Reports CS-93-119*, 1993

[12] J. Carter, J. Bennet and W. Zwaenepoel. “Implementation and Performance of Munin” *Proceedings of the 13th ACM symposium on Operating systems principles* 1991, pp 152-164

[13] P. Keleher, S. Dwarkadas, A. L. Cox and W. Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems” *Proceedings of the Winter 94 Usenix Conference* 1994, pp. 115-131

[14] D. Scales, K. Gharachorloo and C. Thekkath. “Shasta: a low overhead, software-only approach for supporting fine-grain shared memory” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* 1996, pp 174 - 185

[15] J. Carter, A. Ranganathan, and S. Susarla. “Khazana: An Infrastructure for Building Distributed Services.” *Proceedings of the 18th Annual International Conference on Distributed Computing Systems* 1998, pp. 562-571

[16] S. Susarla, A. Ranganathan, and J. Carter. “Experience Using a Globally Shared State Abstraction to Support Distributed Applications” *University of Utah, Department of Computer Science, Technical Report UU-CS-98-016*, August 1998

[17] R. Eskicioglu, A. Marsland, W. Hu, W. Shi. “Evaluation of the JIAJIA software DSM system on high performance computer architectures” *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences* Vol. 8, pp 10, January 1999

[18] L. Iftode, J. P. Singh and K. Li. “Scope consistency: a bridge between release consistency and entry consistency” *Proceedings of the 8th annual ACM symposium on Parallel algorithms and architectures* 1996, pp 277-287

[19] W. Hu, W.Shi and Z. Tang. “JIAJIA: A DSM System Based on A New Cache Coherence Protocol” *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe* 1999, pp. 463-472

[20] E. Jul, H. Levy, N. Hutchinson, A. Black. “Fine-grained mobility in the Emerald system” *ACM Transactions on Computer Systems* Vol. 6, Issue 1, pp 109-133, February 1988

[21] J. Chase, F. Amador, E. Lazowska, H. Levy and R. Littlefield. “The Amber system: parallel programming on a network of multiprocessors” *Proceedings of the 12th ACM symposium on Operating systems principles* 1989, pp 147-158

[22] B. Bershad, E. Lazowska and H. Levy. “PRESTO: A system for Object-oriented Parallel Programming” *Software - Practice and Experience* Vol. 18, Issue 8, pp 713-173, August 1988

[23] H. Bal, F. Kaashoek, and A. Tanenbaum. “Orca: A Language for Parallel Programming of Distributed Systems” *IEEE Transactions on Software Engineering* Vol. 18, Issue 3, pp. 190-205, March 1992

[24] K. Johnson, F. Kaashoek and D. Wallach. “CRL: high-performance all-software distributed shared memory” *Proceedings of the 15th ACM symposium on Operating systems principles* 1995, pp 213-226

[25] Object Management Group. “Common Object Request Broker Architecture (CORBA)”, v3.0

[26] Jan Kleindienst, František Plášil and Petr Tuma. “Lessons learned from implementing the CORBA persistent object service” *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* 1996, pp 150-167

[27] The Open Group. “DCE Portal” http://www.opengroup.org/dce/ 2004-07-22

[28] A. Schill and M. Mock. “DC++: distributed object-oriented system support on top of OSF DCE” *Distributed Systems Engineering* Vol. 1, Issue 2, pp 112-125, December 1993

[29] Sun Microsystems Inc. “Java Remote Method Invocation - Distributed Computing for Java” http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html 2004-08-12

[30] A. Bakker, I. Kuz, M. van Steen, A.S. Tanenbaum and P. Verkaik. “Design and Implementation of the Globe Middleware” *Technical Report IR-CS-003*, June 2003

[31] Sun Microsystems Inc. “Jini Network Technology” http://wwws.sun.com/software/jini/

[32] N. Carriero and D. Gelernter. “Linda in context” *Communications of the ACM* Vol. 32, Issue 4, 1989, pp 444-458

[33] B.-H. Yu, Z. Huang, S. Cranefield and M. Purvis. “Homeless and home-based Lazy Release Consistency protocols on Distributed Shared Memory” *Proceedings of the 27th conference on Australasian computer science* 2004 Vol. 26, pp 117-123

[34] International Business Machines Corp. “TSpaces - Computer Science Research at Almaden” http://www.almaden.ibm.com/cs/TSpaces/

[35] Gigaspaces Technologies Ltd. “The GigaSpaces Cluster” White Paper, May 2004

[36] Gigaspaces Technologies Ltd. “GigaSpaces Distributed Caching” White Paper, 2004

[37] Oracle Corporation. “Oracle® Database Application Developer's Guide - Fundamentals” Part Number B10795-01

[38] Oracle Corporation. “Oracle® Database Application Developer's Guide - Object-Relational Features” Part Number B10799-01

[39] Oracle Corporation. “Oracle® Database Advanced Replication” Part Number B10732-01

[40] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel. “An evaluation of software-based release consistent protocols” *Journal of Parallel and Distributed Computing* Vol. 29, Issue 2, pp 126-141, September 1995

[41] Sun Microsystems Inc. “Collections” http://java.sun.com/docs/books/tutorial/collections/

[42] Microsoft Corp. “System.Collections” http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp

[43] Y. Mirza. “A Compositional Collections Component Framework” *7th International Workshop on Component-Oriented Programming* 2002

[44] R. Raj. “The Active Collections Framework” *ACM SIGAPP Applied Computing Review* 1999, Vol. 7, Issue 1, pp 9-13

[45] Sun Microsystems Inc. “JDBC Technology” http://java.sun.com/products/jdbc/

[46] Sun Microsystems Inc. “JDBC™ Database Access” http://java.sun.com/docs/books/tutorial/jdbc/

[47] Sun Microsystems Inc. “JDBC Documentation” http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/

[48] Sun Microsystems Inc. “Java Data Objects (JDO)” http://java.sun.com/products/jdo/index.jsp

[49] wikipedia.com. “Gnutella - Wikipedia, the free encyclopedia” http://en.wikipedia.org/wiki/Gnutella

[50] I. Clarke. “The Freenet Project - ngrouting - beginner” http://freenet.sourceforge.net/index.php?page=ngrouting

[51] wikipedia.com. “Napster - Wikipedia, the free encyclopedia” http://en.wikipedia.org/wiki/Napster

[52] gnutella2.com. “g2dn: Gnutella2 Developers' Network” http://www.gnutella2.com/

[53] The P-Grid Consortium. “P-Grid - The Grid of Peers” http://www.p-grid.org/

[54] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek and H. Balakrishnan. “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications” *IEEE/ACM Transactions on Networking* 2003, Vol. 11, Issue 1, pp 17-32

[55] L. Onana Alima, S. El-Ansary, P. Brand and S. Haridi. “DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications” Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid 2003 pp 344-350

[56] J. Douceur. “The Sybil Attack” *Revised Papers from the First International Workshop on Peer-to-Peer Systems* 2002, pp 251-260

[57] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz and I. Stoica. “Towards a Common API for Structured Peer-to-Peer Overlays” *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems* 2003

[58] M. Welsh. “Architecture for highly concurrent server applications” http://www.eecs.harvard.edu/~mdw/proj/seda/

[59] A. Gontmakher and A. Schuster. “Java consistency: Non-operational characterizations for Java memory behavior” *Technical report CS0922* Technion 1997