Conceptual Programming Models of Distributed Systems

by

Christopher Wolfe

Queen's University

Kingston, Ontario, Canada

November 2006

Contents

List of Figures

1 Introduction

In this paper, we review the foundations and state of the art in distributed systems toolkits. We focus our discussion on the the programming models of the toolkits, which define how the programmer conceptualises the structure and operation of an application. Three aspects of the programming models are examined in detail to provide a taxonomy: how communication is performed, what implementation details are concealed from the programmer, and how the organisation or implementation may be selected and modified. Each of these aspects is subdivided into common approaches, and accompanied with examples from research literature or industrial practise.

Distributed systems, in which computers without shared memory interact extensively, present a particularly complex programming environment. Despite their complexity, distributed systems are useful for problems which require high performance, high availability, or in contexts where resources are inherently distributed [8]. For example:

- Distributed systems can improve the performance of some huge distributed problems, as demonstrated by SETI@Home [1], Folding@home [52] and distributed rendering environments [57, 39].

- Enterprise databases such as IBM DB2 and Oracle use distribution to improve availability and performance [62].

- Collaborative groupware (e.g. GroupKit [73]) and specialised servers (e.g. in AmoebaOS [79]) use distribution to allow physically separated people and resources to interact.

- All of these benefits appear in environments like massively multiplayer online games, which combine expensive computations, high availability demands and distributed users (e.g. World of Warcraft [19]).

The modern proliferation of interconnected computing devices, such as smart phones, laptops and PDAs, has kindled increased interest in distributed systems [48]. Such devices typically have limited performance and resources, so cooperation between devices is important. Intermittent connectivity, mobility, and problems such as limited batteries motivate the need for adaptability and fault tolerance. Inherent distribution appears in collaborative scenarios, and when a single user is switching between devices or accessing remote resources. Beyond the requirements of distributed systems, the sheer heterogeneity of these devices – e.g. varying processors, displays, input devices – makes developing software for them difficult.

The advantages of distributed systems have led to significant research focused on reducing the complexity of their implementation. *Distributed systems toolkits* seek to reduce complexity by concealing elements of the system, typically including low-level details, like the type of network used to connect the distributed system's nodes, or the device upon which an operation occurs. Each of these toolkits defines one or more programming models – conceptual frameworks that provides the application developer with a simplified view of the system – that are amenable to programming distributed systems. Many toolkits can be customised to interact better with a particular application, for example by introducing custom network protocols or locking algorithms. Unfortunately, these customizations typically require specialized knowledge of the toolkit.

Distributed systems programming models must compromise between complete transparency, where the entire system appears to be one statically configured computer, and no transparency, where the developer must control all details of the distributed communication. In general, and particularly when heterogeneous mobile devices are involved, dealing with all of the complexities would make developing a large, flexible application infeasible. Many toolkits provide a highly transparent model for general application programming and additional mechanisms to access and control the underlying implementation.

It is often useful to represent change in a distributed system model, for example as new users enter a collaborative session or when a computer fails. One of the considerations of a programming model is how changes in the environment can be indicated to the programmer, and how the behaviour of the program can adapt in response. Many traditional toolkits are primarily intended for computational problems in static managed environments [29]. Any change, including a network failing or a new computer being introduced, would require either explicit handling by the application or restarting the system. This restriction is obviously unacceptable for unmanaged systems, for example unrelated devices connected via the Internet, so further research has dealt with toolkits that can change the distributed structure of a running application. In some cases, this *evolution* extends even to dynamically replacing and reconfiguring parts of the toolkit infrastructure.

Research in distributed systems has originated from numerous communities, including databases, mathematical computation, and computer-supported cooperative work (CSCW). As such, it is no surprise that papers use widely varying terminology. For consistency, we impose a single notation and terminology to compare the architectures, advantages and limitations of the various distributed systems toolkits. Section 2 defines our terminology, while section 3 defines our graphical notation.

Our discussion of distributed system toolkits is divided across the three aspects: Communication models, in section 4, deal with how the conceptual behaviour of the system is exposed to an application programmer. Transparencies, in section 5, define particular aspects of the distribution that are hidden by the programming model. Finally, configuration, in section 6, details how the connectivity and implementation of the system is expressed.

2 Terminology

We now define the terminology used throughout this paper. Some of the papers cited here use these terms in substantially different ways. Where terms are needed only in a single section, they will be defined therein.

A *node* refers to a single computational resource, which may have multiple processors and locally shared memory. There will generally be one node per physical computer, though creating multiple *virtual nodes* on one computer is often possible. A *process* is the basic physical unit of concurrency on a node. A single process may have units of internal concurrency, which are identified as *threads*. Both processes and threads are local to a single computer, so may not migrate through the distributed system. *Logical threads* are the unit of concurrency represented in the distributed systems toolkit. In some toolkits, logical threads may migrate between nodes.

A *value* is anything that a variable may contain, usually a record, object or reference. A *record* is an aggregate of one or more values, as in Pascal or a struct in C. An *object* is a value with an associated set of operations, for example an integer or a representation of a graphical window. A *reference* provides a mapping to another value. *Local references* are limited to indicating values present in their node, while *remote references* may cross node boundaries. A *component* is the value or collection of values, potentially with associated threads, that serves as the basic unit of distribution. A *connection* is a communication channel between two or more components.

A *concurrency control* manages parallel access to a resource or set of resources, for example by only allowing one logical thread at a time to execute operations on a value. *Consistency maintenance* deals with keeping two or more values in similar states, for example ensuring distant collaborating users have the same view of a whiteboard. An algorithm used in consistency maintenance is a *consistency protocol.*

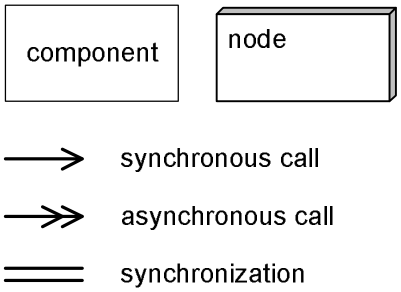


Figure 1: Elements from the Workspace notation

3 Graphical Notation

Through much of this paper we discuss the architecture and evolution of distributed systems. To represent these structures in a precise visual format, we use a subset of the Workspace Architecture [70] conceptual level notation. The particular elements present in this paper are shown in Figure 1 and described below.

Workspace conceptual architectures consist of components interacting via connectors. Each connector may be explicitly anchored to a node, and will then be drawn within the node box. Components may also be floating, in which case they are drawn outside any node. Floating components may be implemented on any node, including those shown in the diagram.

Communication between components is performed only via explicit connectors. The three types of connector each support a different mode of communication, but may all cross node boundaries. Call connectors provide synchronous calls to a single target, similar to imperative method calls. Subscription connectors deliver asynchronous events to zero or more targets. Synchronisation connectors ensure that two or more components maintain a degree of observational equivalence.

When depicting a change in the architecture, we show separate diagrams for the local configurations before and after the evolution.

4 Communication Models

The programming model exposed by a distributed systems toolkit must provide some mechanism allowing cooperating components to communicate. Such a model allows an application programmer to express interaction and concurrency, often without being tied to a precise implementation: for example, most computer networks are based on message passing, but a communication model based on shared data is generally considered easier to program.

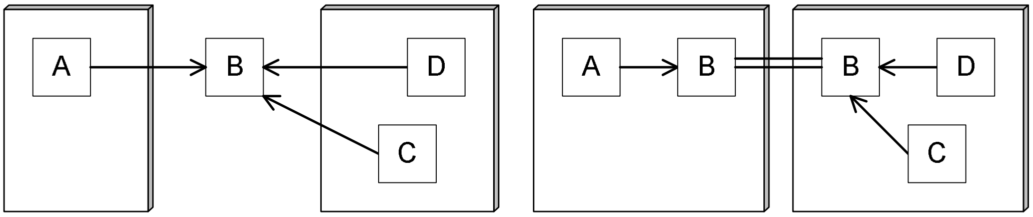
We use four broad categories to describe how communication and distribution are presented conceptually to an application programmer:

- Shared data, in which values appear to be directly accessible from multiple nodes;
- Message passing, where asynchronous events are passed between components;
- Remote procedure call, which allows synchronous method calls between distributed components; and
- Mobile agents, where the components themselves move between nodes.

Toolkits often offer features from more than one of these categories, because many algorithms are easier to describe using a particular communication model [5].

4.1 Shared Data

The shared data model draws from the long history of shared-memory multiprocessor computers. At its core lies the illusion that all of the different computers involved in the distributed system share some physical memory. Conceptually, this behaviour is provided by placing the shared values in a virtual memory area outside any node (Figure 2(a)), or by replicating them across multiple nodes (Figure 2(b)). Shared data models fall into three primary categories: unstructured, record-structured and object-structured.



(a) Single instance at unknown location (b) Replicated instances at multiple locations

Figure 2: Conceptual views of Shared Data. Component B appears to be directly accessible from any node.

4.1.1 Unstructured shared data

This approach to shared data applies consistency and concurrency protocols to raw memory or memory pages [42]. It is most easily pictured as a form of virtual memory in which the data may be stored on a remote computer rather than on a local disk [53].

While this model is more common in environments without a reliance on objects, some object-based toolkits take advantage of its simplicity. Toolkits within this model differ mostly in their approach to allocating memory addresses and managing concurrent access.

Panda [3] is one of the few available examples of an object-oriented unstructured shared data system. It provides a preprocessor and class library for C++, plus an operating system kernel to provide the actual sharing of data. The preprocessor and class library primarily provide features to simplify the user language, including a simple object lookup service and mutual exclusion synchronisation. The kernel divides the address space into a set of static partitions, with at least one public and private region per node. Code on one node accessing memory in a region belonging to another node triggers the kernel to copy values between nodes or migrate logical threads.

As evidenced by Panda and similar non-object-oriented systems, implementing unstructured shared data is relatively simple. Operating system techniques commonly applied to virtual memory and local shared memory can be recycled to provide a classical programming environment. The primary weakness of these systems is the difficulty of identifying or expressing a high-level component model in terms of raw memory. This limitation means that many concurrency and consistency maintenance protocols are hard to provide.

4.1.2 Record-structured shared data

This approach organises the shared data into records, allowing the identification and separate treatment of individual fields. The absence of methods associated with the records maintains much of the simplicity of unstructured shared data, while making use of more information available from the application programming language. This model appears in distributed relational databases, but is most directly applicable to large-scale software in systems like Linda [32].

Linda appeared in the mid eighties as a model for communication and coordination of parallel processes [35]. It provides a collection of primitives that operate on a shared *tuple spaces*. A tuple space in Linda is a shared data construct containing arbitrary records, like a blackboard. The operations for accessing or extracting tuples from the space specify a pattern containing the type or value of each field in the records to be retrieved.

While the Linda primitives are not inherently object-based, they interact well with object-based application languages. Similar efforts like WCL [86] have been implemented for object-based languages. Extensions of Linda have added support for varying methods of distributing the tuples [74] and asynchronous access to the tuple space [30].

The elegant semantics and simplicity of tuple spaces, and similar record-structured shared data, makes them attractive for analysis and implementation. Unfortunately, they behave differently from the usual constructs found in imperative object-oriented programming, so do not integrate cleanly with such programming languages.

4.1.3 Object-structured shared data

This approach organises shared data into objects, each defined by a representation and set of methods. This model follows that of traditional imperative object-oriented programming languages, so combines well with such languages. Many toolkits allow manipulation of the representation only from the associated methods, unlike programming languages like C++ and Java where fields may be non-private. Such a limitation means that the set of possible operations can be easily identified.

Orca [9] provides a good example of a custom language and runtime environment for object-structured shared data. The language supports both sequential and distributed constructs, while maintaining simple behaviour. Shared data is organised into *abstract data types*: types of values that define an explicit set of permissible operations. The actual communication implementation depends on the particular Orca runtime, for example the original paper discusses implementations based on both point-to-point and broadcast messages. Later work added support for replication [6], compile-time analysis [4], and limited fault tolerance [45].

J-Orchestra [84] is a modern example of object-structured shared data based on an existing language. The original work focused on distributing a compiled Java program across nodes using bytecode modification of the application [80]. For example, many Java system classes make use of native resources and executable code. The resulting toolkit is capable of producing a distributed prototype of a Java application based on only the compiled bytecode. More recent variations have explored implementations of the system based on aspect-oriented programming [83], alternatives to Java RMI [81], prototyping for ubiquitous computing [55], and distributed threads [82].

4.1.4 Analysis

Shared data models focus on making the same data seamlessly accessible from multiple nodes. This focus blends together multiple nodes from the programmer's perspective, presenting both benefits and drawbacks. On the positive side, the programmer may ignore node divisions and allow the toolkit free reign to optimize the application. Unfortunately the toolkit is not always capable of producing acceptable behaviour, and the blending can make it difficult for a programmer to isolate and fix problems.

Algorithms and programming techniques developed for concurrent systems generally have to deal with issues of non-uniform memory access (NUMA). Many of these techniques remain applicable in distributed shared data systems, though throughputs and latencies vary far more. Some early shared data toolkits were even designed with the objective of allowing networks of computers to be programmed as a single multi-processor (e.g. Amber [26]).

The high level expression of behaviour provided by a shared data model would theoretically be ideal for optimisation by a toolkit. Automatic compile- and run-time tuning have been provided by a variety of tools (e.g. in Orca [4] and Pangaea [76]). While the gains are significant, they still can not guarantee good behaviour without programmer intervention [55]. Further, the complete node abstraction implied by the shared data model may make it more difficult to isolate and correct situations that the toolkit can not optimise.

In a large distributed system, particularly one involving mobile nodes and unreliable connections, partial failure is a major problem. Dealing with these failures in a shared data system is particularly hard, because the conceptual model conceals any distribution and replication of values is concealed by the model. Not only may groups of values spontaneously disappear, but the groups may vary as the toolkit rearranges the distribution.

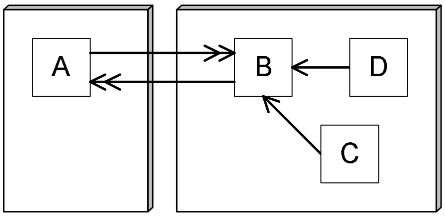


Figure 3: Conceptual view of Message Passing. Communication between components A and B must be via asynchronous message.

4.2 Message Passing

The message passing programming model provides asynchronous communication between components: a message is transmitted from sender to receiver without waiting for a reply, as shown in Figure 3. As a result, this model is easy to implement on common networks, but application built on a message passing system often require algorithms different from those used in sequential programming environments [67]. Message passing models may be categorised based on their queueing model, threading model and read mechanism.

When a message arrives at a component that is not ready, the message will generally be placed in a queue to await processing. The queueing model defines the order in which these messages will be delivered, and how much control the receiver has. The simplest approach makes use of a FIFO queue, processing messages in exactly the order they were delivered (e.g. DACIA [56]). More flexible models can allow messages to be prioritised, discarded or deferred (e.g. Hybrid [63]).

The threading model describes how threads are allocated to process messages, and how concurrent threads are organised. The simplest threading models assign a single thread to each node, allowing only one message to be in processing at a time (e.g. Charm [47]). A more common and powerful approach is to assign a single thread to each object (e.g. DACIA [56] and Thal [50]). These approaches are commonly described as static threading models, because the number of threads does not change in response to messages between existing components. Dynamic threading models, on the other hand, may create threads in response to messages. A simple, though inefficient, dynamic threading model is to begin processing every message immediately when it is received. More flexible approaches provide controls on the object to determine which messages may be processed concurrently, for example via reader/write locks as discussed in Thal [50].

The read mechanism determines how the component code receives messages. Many systems simply invoke a method on the component, as indicated in the message, when the message is processed (e.g. Cool [25] and DACIA [56]). Other approaches permit an executing method to wait for a message with particular properties to arrive (e.g. Active Objects [63] and Thal [50]).

4.2.1 Charm++

Charm++ [47] provides a variant of C++ that includes constructs for specifying concurrency, asynchronous communication, and replication. Charm++ follows from research begun in the late eighties on the Charm Parallel Programming System [46]. Charm and Charm++ provide similar conceptual models, but Charm++ is more completely object-oriented. The design of Charm++ focuses on providing dynamic load balancing and latency tolerance, while striking a balance between unnecessary complexity and hiding performance costs from the programmer.

Programs in Charm++ are built around actors. Each actors is a single object with an associated logical thread, and one or more entry point functions. Messages received for the actor are queued and delivered by executing the appropriate entry point function with a record parameter. To allow distributed algorithms to be expressed more simply and efficiently, Charm++ also provides some shared data constructs.

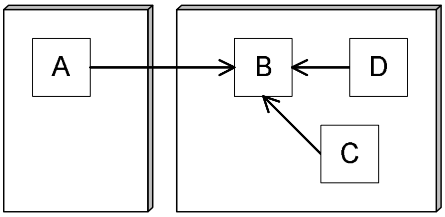


Figure 4: Conceptual view of Remote Procedure Call. `A` may make a call on `B` as if it were a local component.

4.2.2 Analysis

The selection of pure message passing toolkits that use an object-oriented structure is limited relative to those that take a procedural approaches. In many cases (e.g. Agent TCL [38] and PVM [34]), message passing is provided as a secondary model, augmenting one of the other communication models. This appears to be related to the difficulty of expressing common synchronous constructs, like method return values, in a purely asynchronous model.

4.3 Remote Procedure Call

Remote procedure call (RPC) models attempt to emulate local method invocations: the logical thread that performs a remote invocation waits until it completes or fails. Conceptually, method invocations cross node boundaries as synchronous calls, as shown in Figure 4.

4.3.1 Java RMI

Java RMI [78] provides remote invocation of Java objects as part of the standard class library. Almost all of the semantics of local method calls are maintained in Java RMI, though a few serious differences are introduced. The most severe appears in the behaviour of Java's built-in `synchronization` primitive: it locks the local proxy rather than the central object, producing the possibility of distribution-dependant faults. Communication in Java RMI is strictly synchronous: the caller of a method will always wait for it to return before continuing.

As in local Java, logical threads are created explicitly via instances of `java.lang.Thread`. The logical threads may cross node boundaries during method calls. An exception is thrown to the caller of a remote method if the invocation fails, either totally or partially. While Java RMI is a relatively low-level toolkit with minimal transparency, it serves as the foundation for more powerful systems like J-Orchestra [80] and JavaParty [69].

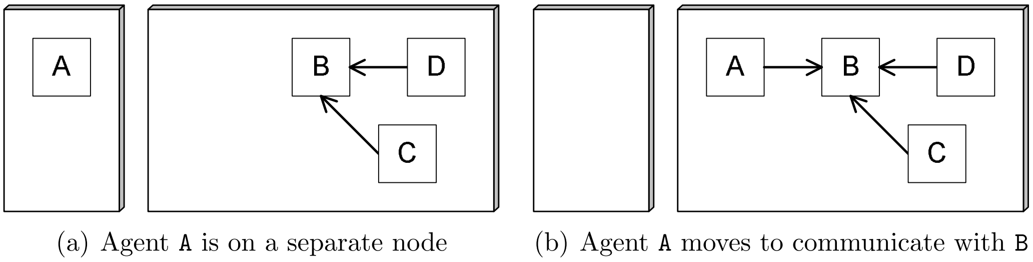
4.3.2 Analysis

RPC models are most obviously useful when a distributed systems toolkit has been designed around an existing imperative programming language. Some common systems, like Java RMI, treat invocations that may be remote differently from those that must be local. Others, like J-Orchestra [80], endeavour to make the local and remote objects indistinguishable.

In some cases, object-structured shared data models could also be described as remote procedure call. The primary difference is what portion of the system conceptually crosses node boundaries: in shared data the values are accessible directly, while in RPC the logical thread moves to the remote node. This difference is primarily relevant to the programmer's expectations of performance and partial failure, as modification to the shared data would not conceptually occur until it was available locally.

4.4 Mobile Agents

The mobile agent programming model focuses on the movement of *agents* throughout the system. When an agent needs to acquire information from a value that not present on its local node (Figure 5(a)) it moves to the same node as the value (Figure 5(b)).



(a) Agent `A` is on a separate node (b) Agent `A` moves to communicate with `B`

Figure 5: Conceptual views of Mobile Agents. Component `A` must move to the same node as `B` to communicate with it.

Each agent is conceptually a logical thread with aggregated values. In a pure agent system, the agents may access only data in their local node. Computational algorithms in this model are often similar to the equivalent shared data algorithm [67], making them substantially simpler than message passing. Some agent models specify all moves explicitly (e.g. DSC [67] and MESSENGERS [33]), while others move the agents automatically in response to data access (e.g. Obliq [23]) or node availability.

Mobile agent models with implicit movements will often appear similar to remote procedure call and shared data models. The distinguishing factor is again based on the conceptual node crossing: mobile agents relocate a logical thread and associated data across a node boundary, but do not necessarily return to the originating node.

4.4.1 Obliq

Obliq [23] provides an elegant mobile agent programming model, that also allows remote procedure calls to be represented. Communication between nodes takes the form of *closures*, which combine code to execute with the values it requires. In this model, remote procedure calls become simply a closure where the only free values are parameters or global variables.

Actors are created through explicit language features, while migration is performed implicitly. Obliq's relatively simple language can simple represent a surprising variety of distributed algorithms [23].

4.4.2 Analysis

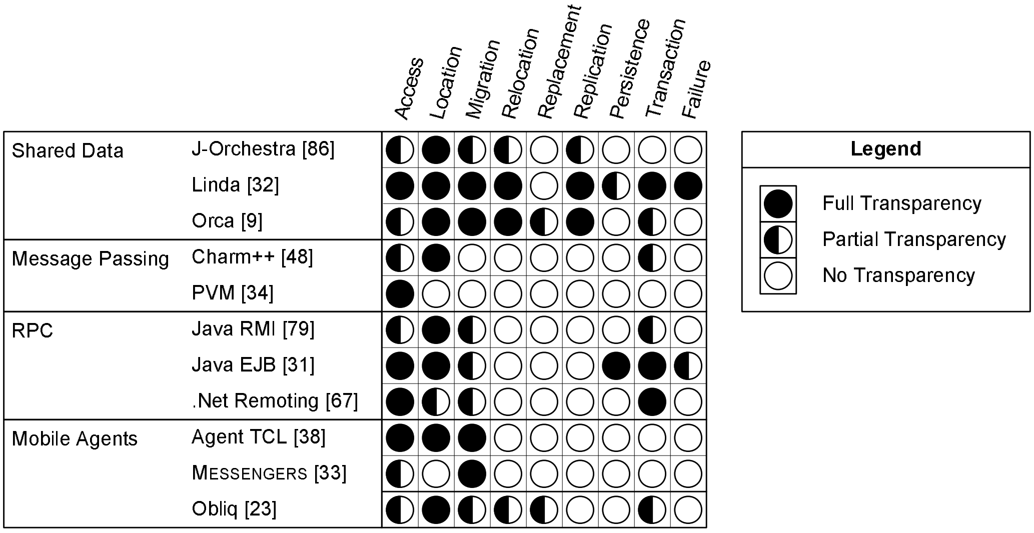
Mobile agent toolkits represent a compromise between message passing and shared data models. In general a component must specify the nodes to which it will move, though some systems allow querying the location of another component or automatic movement when accessing a remote value. This means that while the application programmer is not always forced to consider node boundaries, they are visible in the conceptual model. As a result, the mobile agents are more complex and flexible than shared data, but far more amenable to traditional algorithms and architectures than message passing.

Exposing nodes to the programmer limits the operations that may be performed by the toolkit, removing some of the flexibility possible in other models and requiring the programmer to be aware of any changes. For example, in a system with both desktop and handheld nodes, an application would need to carefully avoid moving computationally-expensive tasks to the handhelds. To reduce this problem, some toolkits introduce *virtual nodes*: clusters of values to which an agent may migrate without knowing the physical node upon which they reside (e.g. Dejay [21]). This approach, unfortunately, raises the same diagnostic and failure problems found in shared data models.

Research in mobile agent-based toolkits is relatively recent, and has not yet reached a wide industrial audience. As the model compares well with others [67], this appears to be related to a lack of commercial toolkits and experience.

5 Distribution Transparencies

The complexity of distributed systems, particularly when they must change over time, leads us to an important property of toolkits: what aspects of distributed architecture and evolution may be ignored by an application programmer. These aspects may be present



		Access	Location	Migration	Relocation	Replacement	Replication	Persistence	Transaction	Failure
Shared Data	J-Orchestra [86]	◐	●	◐	◐	○	◐	○	○	○
	Linda [32]	●	●	●	●	○	●	◐	●	●
	Orca [9]	◐	●	●	●	◐	●	○	◐	○
Message Passing	Charm++ [48]	◐	●	○	○	○	○	○	◐	○
	PVM [34]	●	○	○	○	○	○	○	○	○
RPC	Java RMI [79]	◐	●	◐	○	○	○	○	◐	○
	Java EJB [31]	●	●	◐	○	○	○	●	●	◐
	.Net Remoting [67]	●	◐	◐	○	○	○	○	●	○
Mobile Agents	Agent TCL [38]	●	●	●	○	○	○	○	○	○
	MESSENGERS [33]	◐	○	●	○	○	○	○	○	○
	Obliq [23]	◐	●	◐	◐	◐	○	○	◐	○

Legend	
●	Full Transparency
◐	Partial Transparency
○	No Transparency

Figure 6: Distribution transparencies in selected toolkits

in any communication model, though particular forms of model are more ameanable to particular transparences: for example, shared data communication models inherently provide location transparency.

The ISO Reference Model for Open Distributed Processing (RM-ODP) defines distribution transparencies, aspects that are important in distributed systems and may be useful to conceal or abstract [43]. The following transparencies, except for replacement, are based on those specified in RM-ODP. Replacement transparency is added to include general configuration techniques in the wider family of distributed systems evolutions.

The degree to which each of the transparencies is present in a particular distributed systems toolkit hints at the level of separation between the application programmer and the underlying implementation. Figure 6 shows a summary of the transparencies provided by selected toolkits discussed in this paper.

It is often possible to provide functionality equivalent to the transparency within a toolkit. For example, both Java RMI and .Net Remoting could reasonable be extended to support migration transparency, by replacing parts of their infrastructures.

Some transparences tend to be associated, for example location and migration transparencies. These tendencies appear to reflect the intent of the developers, rather than a necessary dependency.

5.1 Access Transparency

Given the variety of modern computing devices, allowing the interaction of fundamentally different nodes is particularly important. Access transparency conceals the differences between heterogeneous nodes: e.g. different operating systems, data representations, display devices, machine languages. Some forms of access transparency focus on local capabilities, for example the TrollTech Qt library provides cross-platform access to user interface and networking facilities. Other forms of access transparency focus on the interaction between nodes: e.g. SOAP provides a uniform data representation and communication protocol, and the Java virtual machine [54] allows the same bytecode to be executed on any system.

Access transparency is particularly useful in large, unmanaged distributed systems or those involving lightweight devices. In a large system, the expense of maintaining completely homogeneous nodes or maintaining new implementations for each type of node may be prohibitive: consider implementing a web server if every operating system used an incompatible HTTP protocol. Lightweight and embedded devices provide a wider range of hardware and capabilities than are typically found in desktops, so exasperate the problem of multiple implementations.

5.2 Location Transparency

Location transparency hides the nodes upon which other values reside, both during an initial lookup and later interaction. The objective is that the application does not need to specify the node upon which a component can be found, as shown in Figure 7. For

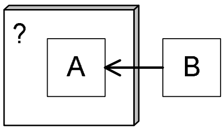


Figure 7: Location transparency. Component `B` does not need to know which node `A` exists on.

example, Microsoft .Net Remoting [65] allows remote values to be treated as local references, concealing the address of the remote node and any necessary network connections. The C language binding of Sun RPC [77] provides very little location transparency, as connections are explicitly created by server address and remote references are not supported.

Traditional techniques for providing location-transparent lookup have involved either centralised or fully replicated repositories [27]. Structured distributed repositories appear where the identifiers of objects can be decomposed in a hierarchal fashion, for example in DNS names. A more recently popularised technique is the use of distributed hash tables (DHTs): the actual location of each object is stored on a subset of the nodes determined by hashing a well-known property of the object (often the name or a unique identifier) [10]. When the object is moved, only the responsible subset need to be notified.

Location transparency serves to separate the application logic from the physical distribution, so proves useful in large or dynamic environments. Web browsers demonstrate the advantage of location transparency: rather than require the user to memorise IP addresses, a DNS lookup is used to resolve human-readable domain names.

5.3 Migration Transparency

While a distributed system is running, it can become useful to migrate a value from one node to another as shown in Figure 8. This allows the system to adapt in response to events like a changing environment or user requests. For example, a word processor could be moved from a laptop to a desktop when the user wishes to use the larger display.

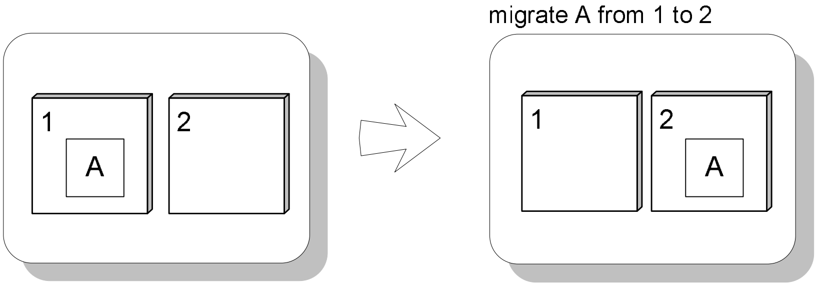


Figure 8: Migration transparency. Component `A` may move transparently between nodes.

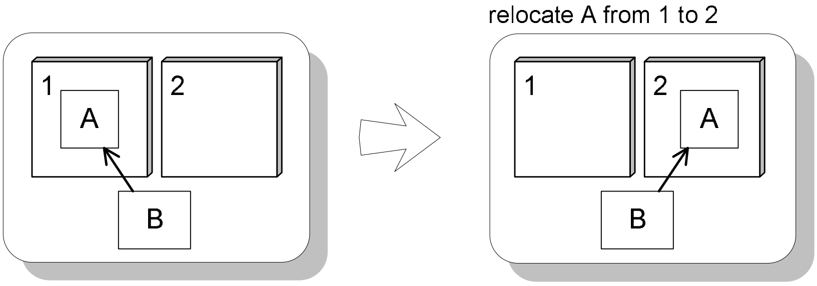


Figure 9: Relocation transparency. Component `A` may move between nodes without breaking `B`'s connection to it.

Migration transparency hides from a value or logical thread the fact that it has moved between nodes. While the application programmer may need to perform an operation to trigger the move, it would not be necessary to embed special logic in the value to be moved. Partial migration transparency may require some special logic, for example handling marshaling or suspend/resume in application code [71].

Most toolkits that provide migration transparency will automatically maintain outgoing connections from the moving component (e.g. Emerald [44] and Orca [6]). If the system also provides location transparency, it will need to account for the movements.

5.4 Relocation Transparency

When a value is migrated between nodes as shown in Figure 9, other values that hold references to it will need some means to communicate with its new position. Relocation transparency hides from a value or logical thread the fact that another value has

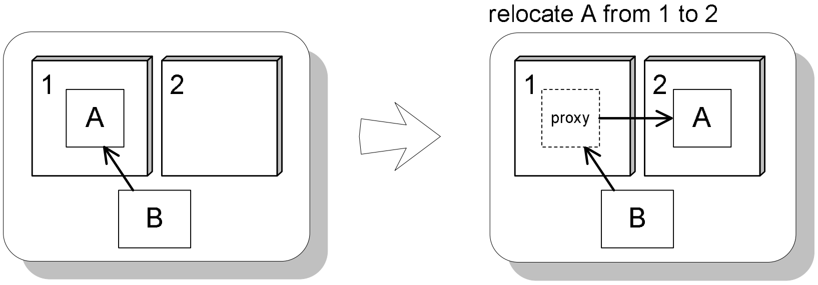


Figure 10: Relocation using the breadcrumbs technique. Rather than update `B`'s connector, a proxy is created at `A`'s old location.

moved between nodes. The move may have been explicitly requested by the application programmer, but housekeeping operations such as updating references or redirecting messages should be automatic.

A variety of techniques are commonly used to implement relocations, which may be visible when the toolkit provides incomplete relocation transparency. Conceptually, relocation consists of suspending all operations on the object to be moved, moving the object, updating all references to it, and then resuming operations.

The conceptual algorithm could be implemented directly, but imposes a substantial cost when moving objects with many connections. A simple technique used to avoid updating references is leaving a proxy object at each past location of a mobile object (see Figure 10). The primary drawback of such a trail is the large number of proxies that may be required to trace a highly mobile object, as in Obliq [23].

Relocation transparency is most commonly found in combination with location and migration transparency, but can be useful independently. For example, the contents of a document could be marshaled and copied from a laptop to a desktop while still allowing the word processor on the laptop to change the document.

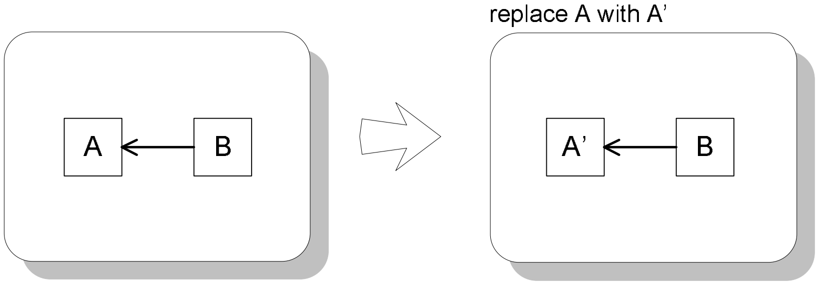


Figure 11: Replacement transparency. `A` may be replaced with a new component `A'` without breaking connections to it.

5.5 Replacement Transparency

Software updates or a changing environment may make it necessary to replace the implementation of a value, as sketched in Figure 11. Such modifications in a traditional software package, for example applying security patches to Microsoft Windows, could require completely restarting the modified node or group of nodes. Replacement provides the ability to replace the implementation of a value without restarting the distributed system, and potentially without the change being visible to other values or threads.

Replacement transparency hides from a value the fact that another value or group of values have been replaced with a different implementation. This transparency has two primary points of complexity: how references are updated to the new implementation, and how the state of the old and new implementations is synchronised. Reference updates during replacement are very similar to those encountered during relocation, and share many of the implementations.

Maintaining behaviour during value replacement requires the ability to force the new value into a state that observationally equivalent to the old one. While some work has been done in automating such operations, a general solution still requires a programmer to define migration behaviour [66]. For example, an automatic tool might infer that fields with the same names should be copied, but be unable to guess initial values for added fields.

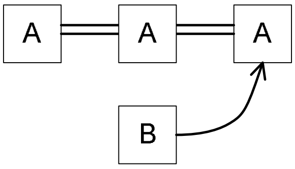


Figure 12: Replication transparency. Component `B` does not need to consider which `A` it is connecting to or interacting with.

Replacing groups of components is necessary for general replacement: for example, to apply an update that refactors a single component into two, or vice versa. This capability was not considered in any of the reviewed systems.

In general, replacement transparency is not dealt with except in research directly approaching the problem of major reconfiguraion or upgrades (e.g. Argus [20] and Djinn [60]), rather than generally usable distributed systems toolkits.

5.6 Replication Transparency

Maintaining multiple replicas of a value, as shown in Figure 12, can provide significant improvements in performance and reliability [12, 7, 51]. For example, a password authentication server could maintain a full copy of a password database to ensure quick response and avoid failure if the database server crashed. Establishing and maintaining observational equivalence across the replicas is, in general, an open research problem.

Replication transparency provides multiple copies of a value that appear to be a single value. Partial replication may be provided without a complete copy, for example through the use of caches (e.g. Clock [37] and Munin [13]), but may still be fully transparent.

Providing this transparency divides into three aspects: creating, maintaining, and accessing the replication [15]. Creating the replication involves duplicating all or part of the value in question, and setting up consistency maintenance. Consistency maintenance keeps the states of the various values the same, within bounds depending on the exact

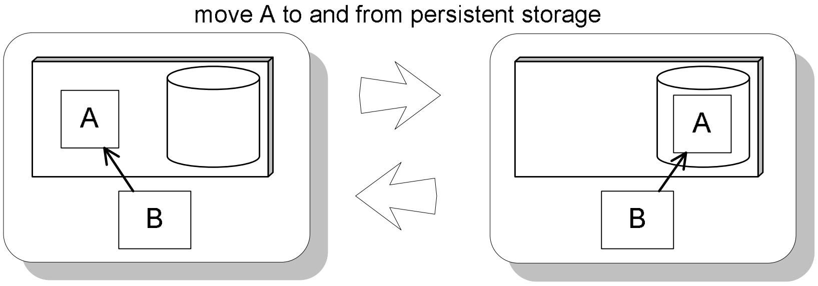


Figure 13: Persistence transparency. Component `A` may transparently move to and from persistent storage, without affecting its state or connections.

concurrency protocol (e.g. The Grid Protocol [28], release consistency [49], and rule-based object coordination [2]). Accessing the replication requires interacting with the correct value, for example by identifying a nearby copy [72].

5.7 Persistence Transparency

Saving and loading values to and from persistent storage is required by almost all applications. These storage formats may be the result of marshaling a graph of values (e.g. Java Serialization), defined by formal standards (e.g. the OpenDocument format [64]), or simply ad-hoc storage formats.

Persistence transparency conceals from a value, and logical threads interacting with the value, its movement or existence in different types of storage. The most common form of persistence transparency is virtual memory provided by an operating system, in which inactive data is paged out to the hard disk. Other approaches could allow operations to be performed directly on the persistent storage, for example to manipulate a stored object without completely loading it.

Applications of persistence transparency include working with data that is too large to fit in main memory or saving information to disk without requiring special code. A wide variety of systems have been developed for storing objects in an entity-relational database, for example the Java Persistence API [31] being developed as part of EJB 3.0.

5.8 Transaction Transparency

When multiple logical threads seek to interact with the same value, or group of values, it is often necessary to restrict their concurrency to prevent unexpected outcomes. For example, to avoid deadlock or prevent inconsistent data being returned from a data structure. Such concurrency control is not an easy problem in a non-distributed system, and is further complicated by replication, inconsistent communication times, and partial failure. A commonly cited example of the importance of transaction handling appears in Massively Multiplayer Online Game (MMOG) shops when multiple players simultaneously try to purchase a unique item. The application must ensure that one, and only one, player receives the item and pays for it.

Concurrency control algorithms serve to reduce the concurrency that must be considered by the application programmer. For example, the Java `synchronized` primitive and .Net apartment threading employ mutual exclusion to ensure only a single thread manipulates an object at a time. More complex algorithms can permit limited concurrency (e.g. read/write lock), or provide a means to resolve conflicts when they occur (e.g. rollback, forward correction).

Transaction transparency conceals the selection, use and behaviour of a transaction algorithm from the application programmer. For complete transparency, this would require the toolkit to enter, commit, abort and restart transactions automatically. Less complete approaches require the programmer to identify the start and end of transactions, and may expose algorithm-specific behaviour. Organising multiple heterogeneous concurrency control algorithms into a single model is relatively difficult, but has been approached by work like Silva et al [75].

5.9 Failure Transparency

In a large or unmanaged distributed system, it is virtually ensured that some portion will fail unexpectedly. Recovering from failure requires identifying the cause of the fault, and evolving the system at runtime to maintain reasonable behaviour. For example, a network connection may fail or a computer may be suddenly powered off.

Failure transparency hides faults, errors and recoveries from the affected values and logical threads. Where it is not possible to completely conceal the failure, it may be indicated to the application in a fashion consistent with the programming model. For example, completely losing network connectivity in a pure mobile agent model would result in the inability of local agents to change nodes.

Because of the complexity of distributed systems, identifying and recovering from arbitrary failure is a fundamentally difficult problem. For example, a TCP/IP network connection that drops might be reestablished automatically, but a tightly-configured firewall may be very difficult to circumvent. Further, the system must decide when to notify the user of severe failures, for example if a remote node in a collaborative editing session is unreachable.

When failure and other transparencies are combined, complex problems can theoretically be concealed. For example, responding to the failure of a remote node by using cached values while switching to replicas of its objects.

6 Configuration

Much of the complexity of a object-based distributed system relates to how components are selected, connected and placed on nodes. The *configuration* of a distributed system deals with this organisation, and how it evolves over time.

Flexible configuration is crucial when devices with different capabilities or environments are present in the system. For example, while a collaborative editing system based on desktops might require Microsoft Word, extending the same requirement to lightweight PDAs would be unacceptable. A better solution would provide a different node configuration for the PDAs, with lower requirements. As the number of different conditions grows, designing a complete node configuration for each possible situation becomes infeasible. Dealing with these situations requires combining many small decisions into a configuration.

Changes over time may also be a motivation for configuration changes. For example a groupware application should support users joining and leaving a collaboration. Other changes are useful in mobile devices, for example CARISMA [22] allows an image processing application to automatically switch between color and black-and-white display depending on available power.

Traditionally, configuration was either performed implicitly or via a configuration API. More recent approaches have introduced dedicated configuration languages, automated configuration, and combinations of the different techniques.

6.1 Implicit Configuration

In some cases, it is possible to describe the distributed behaviour of a program completely in its source code. These languages allow an implicit configuration to be built from the interaction of objects. This approach is most common in special-purpose programming languages for distributed systems (e.g. Obliq [23]), or when legacy programs need to be distributed (e.g. JavaParty [69]).

Implicit configuration based on existing programming languages is often described as intuitive to programmers, because it provides behaviour similar to non-distributed programming. If the programming language is similar, or identical, to common languages it may be possible to retrofit existing applications: distributing a non-distributed application, making it multi-user application, or simplifying an existing distributed application.

The transparency of distribution may also be a drawback, as there is no clear separation between distributed and non-distributed aspects of the application. Furthermore, subtle differences between the local and distributed semantics may create severe errors: both Java RMI and .Net Remoting change the behaviour of object monitors to one which can result in distribution-sensitive deadlocks.

The choice of language in an implicit configuration system may limit the distributed behaviours that can be supported. Common C-like syntax, for example, can not directly express asynchronous messages or synchronisation of objects [1]. Representing either behaviour requires modifying the language or introducing some other form of configuration.

6.2 Configuration API

Rather than integrating distributed behaviours into the syntax of a language, it is possible to specify all distributed configuration via an API or class library. This approach allows an object to explicitly control, or modify, its distributed behaviour without being limited by its programming language [16]. Configuration APIs are typically defined for toolkits that support one or more existing programming languages (e.g. PVM [34]).

Configuration APIs may be easier to learn than implicit configuration because they do not modify the semantics of the programming language, and are simply another library. As a result, a programmer need only deal with the API when implementing the distributed portions of an application. The opposing argument is that minor changes in the programming language are easier to learn than a completely new, and often complex, library.

The primary drawback of configuration APIs is that they often require an awkward syntax for expressing simple constructs. For example, Figure 14 shows a simple data broadcast in PVM.

[1] While object aliasing can provide total identity, it can not represent partial synchronisation or merging of distinct objects.

```
pvm_initsend(PvmDataRaw);
pvm_pkint(&intslen, 1, 1);
pvm_pkint(ints, intslen, 1);
pvm_pkint(&fltslen, 1, 1);
pvm_pkfloat(flts, fltslen, 1);
pvm_mcast(ids, nids, 1);
```

Figure 14: Broadcast of two arrays in PVM from [34]

In highly-dynamic languages, including many scripting languages (e.g. Python), a configuration API may provide proxy objects that can be invoked using the host programming language. This allows a more familiar syntax for manipulating remote objects, but encounters many of the problems of implicit configuration.

Combining implicit and API-based configuration leads to a technique that is typically described as reflection: default distributed behaviour is inferred from programming language semantics, but may be inspected and customised through the use of an API. Reflective middleware has become increasingly popular as a flexible extension of common industrial middleware [18].

6.3 Configuration Language

An alternative to describing the architecture of the system in an implementation language is to define it using an external language. These configuration languages include high-level graphical representations (e.g. ClockWorks [36]), special-purpose languages (e.g. MANIFOLD [68] and Regis [59]), and attributes added to existing languages (e.g. J-Orchestra [83] and Munin [24]).

Configuration languages provide a clean separation between programming the behaviour of a component, and the organisation of multiple components. While this separation decreases the complexity of most tasks, it requires special handling for evolutions triggered by application logic. Unlike implicit configuration, the programming language will not restrict the types of inter-value communication that may be expressed. Further, it may be possible to retrofit existing source code with complex distributed behaviour.

A graphical configuration language provides an environment that is very different from that used in traditional development. With appropriate tool support, a graphical tool may be significantly easier to learn and use than an equivalent textual language [61].

Textual configuration languages, either as special-purpose languages or attributes, tend to be declarative in nature (e.g. Darwin [58]). This approach is generally described as less complex than an imperative configuration language, but may be less familiar to an application programmer.

The primary drawback to a special-purpose configuration language is that it requires an application programmer to learn and use an additional language. Both languages are often required before even a primitive prototype can be tested.

6.4 Automatic Refinement

As the organisation of distributed systems becomes more complex, increasing focus appears on automating changes. For example, many shared data toolkits adapt the distribution of a value based on its usage: some simply by moving the data to the node that used it most, or more flexibly by changing its replication and consistency maintenance algorithm.

Automatic refinement is often combined with one of the other configuration methods, so that a programmer may manually tweak the system. Three common categories of automatic refinements appear in the literature, differentiated by their information sources and effects: static, statistical and structural.

Static refinement takes place before the distributed system is started by extracting likely interaction patterns from the application. This approach draws heavily on techniques used to optimise parallel programs on multi-processors. Banerjee [11] and Hiranandani [40] present samples of such optimizations.

Statistical refinements are performed while the system is running, and are based on various performance and load measurements. The techniques deployed in this area are similar to those found in distributed databases [62]. Common statistical refinements can move groups of values to reduce network communications (e.g. AMPI [14]), or change the replication of a value based on its access pattern (e.g. ADR [85] and Active Harmony [41]). The primary limitation of statistical refinements is that they deal with relatively low-level changes, and are often difficult to apply to application components.

Structural refinements are the most powerful and complex approach to automated refinement. By combining application-level attributes with statistical data gathering, these refinements focus on wide-reaching and complex changes to occur in the running system. For example, CARISMA [22] presents a constraint-based system that switches between colour and black-and-white display modes depending on available battery power.

6.5 Analysis

The two primary problems in configuration is identifying an appropriate improvement, and then applying the necessary configuration changes. Traditional approaches using implicit configuration require that the toolkit be modified to support additional configurations (e.g. Orca [6]). More flexible approaches, most commonly in the middleware sphere, have introduced configuration APIs that allow the implementation details to be customized (e.g. OpenORB [17]). Configuration languages allow a high-level model to be provided, but still differ substantially from the programming language and communication model. Automatic refinement presents a potential solution to these problems, but remains limited and difficult to extend.

These problems suggest that a system should provide a simplified model not only for high-level application programming, but also for configuration. While some aspects of this approach appear in work like by Blair et al [16], the model exposed by reflection remains tied tightly to the particular middleware. Allowing automated refinements to be defined between two communication models, one associated with the high-level model exposed to the application and one tied to the implementation, could drastically simplify the task of controlling application behaviour.

7 Conclusion

Given the wide variety of techniques and systems developed through distributed systems research, it would appear that most variations have already been explored. Indeed, toolkits supporting multiple models, many transparencies and clean high-level semantics are widely available. Industrial uptake has been mostly limited to fairly simple toolkits that integrate with preexisting programming languages.

The weakness of existing solutions appears primarily when dealing with adaptation, which was far less important in traditional distributed systems. Common approaches allow either low-level changes driven by the toolkit, or high-level changes coded explicitly in the application. A few constraint-based approaches show promise in simplifying the high-level changes, but still have trouble expressing wide-spread structural modifications.

The greatest limitation appears to be the vast gap between the conceptual model exposed to the application programmer and the implementation details necessary to express changes. Bridging this gap requires a simple model that can represent the incremental decisions made between the high level conceptual environment and the actual implementation. Given such a model, adaptive refinements could be combined and applied at any level, independent of the actual middleware.

While tremendous research interest has been directed at toolkits for programming distributed systems over the past decades, it is far from a solved problem. Indeed, the popularity of low-powered mobile devices prevents many of the traditional techniques from even being applicable. Automating the adaptation of a flexible architecture demonstrates promise in providing for these systems, but remains limited in capability.

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[2] J. Andreoli, H. Gallaire, and R. Pareschi. Rule-based object coordination. In *ECOOP '94: Selected Papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 1–13, London, UK, 1995. Springer-Verlag.

[3] H. Assenmacher, T. Breitback, P. Hübsch, and V. Scharz. PANDA – supporting distributed programming in C++. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 361–383. Springer-Verlag, 1993.

[4] H. E. Bal and M. F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 162–177, 1993.

[5] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in Orca. In *IEEE CS International Conference on Computer Languages*, pages 79–89, 1990.

[6] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[7] H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency Practice & Experience*, 4(5):337–355, August 1992.

[8] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[9] H. E. Bal and A. S. Tannenbaum. Distributed programming with shared data. In *Proceedings of ICCL — International Conference on Computer Languages*, pages 82–91, Miami, Florida, 1988. IEEE Computer Society Press.

[10] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.

[11] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[12] D. Barbara and H. Garcia-Molina. Replicated data management in mobile environments: Anything new under the sun? In *Applications in Parallel and Distributed Computing*, pages 237–246, 1994.

[13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[14] M. Bhandarkar, L. Kal, E. Sturler, and J. Hoeflinger. Object-based adaptive load balancing for MPI programs. Parallel Processing Laboratory, Technical Report 00-03, September 2000.

[15] K. P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[16] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. A. Saikoski. A principled approach to supporting adaptation

in distributed mobile environments. In N. P. and R. I., editors, *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'2000)*, pages 3–12, Limerick, Ireland, June 10-11 2000. IEEE.

[17] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in OpenORB. In *WOSS '02: Proceedings of the 1st Workshop on Self-Healing Systems*, pages 9–14, New York, NY, USA, 2002. ACM Press.

[18] G. S. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware*, pages 262–267, New York, NY, USA, 2004. ACM Press.

[19] Blizzard Entertainment. *World of Warcraft Surpasses Five Million Customers World-Wide*, December 2005.

[20] T. Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT/LCS/TR-303, Massachusetts Institute of Technology, 1983.

[21] M. Boger, F. Wienberg, and W. Lamersdorf. Dejay: Unifying concurrency and distribution to achieve a distributed Java. In *Proceedings of TOOLS Europe '99*, Nancy, France, June 1999. Prentice Hall.

[22] L. Capra, G. S. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting reflection in mobile computing middleware. *SIGMOBILE Mobile Compututing & Communication Review*, 6(4):34–44, 2002.

[23] L. Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 286–297, 1995.

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

[25] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[26] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.

[27] D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, 1989.

[28] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid Protocol: A high performance scheme for maintaining replicated data. *Knowledge and Data Engineering*, 4(6):582–592, 1992.

[29] R. S. Chin and S. T. Chanson. Distributed, object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, 1991.

[30] N. Davies, A. Friday, S. P. Wade, and G. S. Blair. An asynchronous distributed systems platform for heterogeneous environments. In *EW 8: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 66–73, New York, NY, USA, 1998. ACM Press.

[31] L. DeMichiel and M. Keith. *JSR 2200: Enterprise JavaBeans, Version 3.0, Java Persistence API.* Sun Microsystems, 2005.

[32] A. Douglas, A. Rowstron, and A. Wood. Linda implementation revisited. In P. Nixon, editor, *Proceedings of the 18th World OCCAM and Transputer User Group Meeting*. IOS Press, 1995.

[33] M. Fukuda, L. Bic, M. B. Dillencourt, and F. Merchant. Intra- and inter-object coordination with MESSENGERS. In *Proceedings of the First International Conference on Coordination Languages and Models (COORDINATION '96)*, pages 179–196, London, UK, 1996. Springer-Verlag.

[34] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[35] D. Gelernter. Generative communication in Linda. *ACM Transactions of Programming Languages and Systems*, 7(1):80–112, 1985.

[36] T. Graham, C. Morton, and T. Ursnes. ClockWorks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, 7(2):175–196, 1996.

[37] T. C. N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'96)*, pages 1–10, Seattle, WA, USA, November 1996.

[38] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, Monterey, CA, 1996.

[39] I. J. Grimstead, N. J. Avis, and D. W. Walker. Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment. In *SC*

'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, page 1, Washington, DC, USA, 2004. IEEE Computer Society.

[40] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 86–100, New York, NY, USA, 1991. ACM Press.

[41] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in Active Harmony. *Cluster Computing*, 2(3):195–205, 1999.

[42] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[43] ISO/IEC. *ITU-T Rec. X.901 — ISO/IEC 10746-1 Open Distributed Processing - Reference Model - Part 1: Overview*, 1998.

[44] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[45] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.

[46] L. Kale. Parallel programming with Charm: an overview. Parallel Programming Laboratory, Technical Report PPL-TR-93-8, University of Illinois, Urbana-Champaign, Department of Computing Science, July 1993.

[47] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, New York, NY, USA, 1993. ACM Press.

[48] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17, 1994.

[49] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, 1995.

[50] W. Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois, Urbana-Champaign, 1997.

[51] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[52] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Modern Methods in Computational Biology*, 2003.

[53] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[54] T. Lindholm and F. Yellin. The Java virtual machine specification, 1999.

[55] N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning: A promising approach to prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, July-Sep 2004.

[56] R. Litiu and A. Prakash. DACIA: a mobile component framework for building adaptive distributed applications. *SIGOPS Operating System Review*, 35(2):31–42, 2001.

[57] K. P. C. Madhavan, L. L. Arns, and G. R. Bertoline. A distributed rendering environment for teaching animation and scientific visualization. *IEEE Computer Graphics and Applications*, 25(5):32–38, 2005.

[58] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[59] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, Pittsburgh, March 1994.

[60] S. Mitchell, H. Naguib, G. Coulouris, and T. Kindberg. Dynamically reconfiguring multimedia components: A model-based approach. In *EW 8: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 40–47, New York, NY, USA, 1998. ACM Press.

[61] K. Morton. Tool support for component-based programming. Master's thesis, York University, North York, Canada, 1994.

[62] M. Nicola and M. Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, 2000.

[63] O. M. Nierstrasz. Active objects in Hybrid. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 243–253, New York, NY, USA, 1987. ACM Press.

[64] OASIS. *Open Document Format for Office Applications (OpenDocument) v1.0*, May 2005.

[65] P. Obermeyer and J. Hawkins. Microsoft .NET Remoting: A technical overview. In *MSDN Library*. Microsoft Corporation, July 2001.

[66] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 649–658, Montreal, Quebec, Canada, October 2002.

[67] L. Pan, L. F. Bic, and M. B. Dillencourt. Distributed sequential computing using mobile code: Moving computation to data. In *Proceedings of the 30th International Conference on Parallel Processing (ICPP-01)*, Valencia, Spain, September 2001.

[68] G. A. Papadopoulos. Distributed and parallel systems engineering in MANIFOLD. *Parallel Computing*, 24(7):1137–1160, 1998.

[69] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[70] W. G. Phillips, T. C. N. Graham, and C. Wolfe. A calculus for the refinement and evolution of multi-user mobile applications. In *Proceedings of the Twelfth International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS '05)*, 2005.

[71] G. P. Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility.* PhD thesis, Politecnico Di Torino, Italy, 1998.

[72] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures (SPAA '97)*, pages 311–320, New York, NY, USA, 1997. ACM Press.

[73] M. Roseman and S. Greenberg. GroupKit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 43–50, Toronto, Ontario, 1992. ACM Press.

[74] G. Russello, M. Chaudron, and M. van Steen. Customizable data distribution for shared data spaces. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2003.

[75] A. R. Silva, J. Pereira, and P. Sousa. A framework for heterogeneous concurrency control policies in distributed applications. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD '96)*, page 105, Washington, DC, USA, 1996. IEEE Computer Society.

[76] A. Spiegel. PANGAEA: An automatic distribution front-end for Java. In *Proceedings of the 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 93–99, London, UK, 1999. Springer-Verlag.

[77] Sun Microsystems. *Remote Procedure Call Programming Guide*, 1986.

[78] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.10 edition, 2004.

[79] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[80] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, pages 178–204, London, UK, 2002. Springer-Verlag.

[81] E. Tilevich and Y. Smaragdakis. NRMI: Natural and efficient middleware. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 252, Washington, DC, USA, 2003. IEEE Computer Society.

[82] E. Tilevich and Y. Smaragdakis. Portable and efficient distributed threads for Java. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 478–492, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[83] E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *Proceedings of 2003 Automated Software Engineering (ASE '03)*, 2003.

[84] E. Tilevicha and Y. Smaragdakis. Automatic application partitioning: The j-orchestra approach. In *Proceedings of 8th ECOOP Workshop on Mobile Object Systems*, 2002.

[85] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[86] A. Wood. Coordination with attributes. In *Proceedings of the Third International Conference on Coordination Languages and Models (COORDINATION '99)*, pages 21–36, London, UK, 1999. Springer-Verlag.