Archipelago
Eine Plattform zur Erforschung der Interaktion von Komponenten Verteilter Anwendungen

Archipelago
A Research Platform for Component Interaction in Distributed Applications

by

Eric Seckler

A thesis submitted to the
Hasso Plattner Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in IT-Systems Engineering

Supervisor

Prof. Dr. Robert Hirschfeld

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

January 20, 2015

Abstract

Distributed applications consist of different parts, which interact across distribution boundaries to achieve a common goal. Different interaction tasks require different mechanisms to communicate, coordinate, or share data or computation. As a result, technologies employed to facilitate these tasks are often extended, customized, and combined to fit application-specific needs.

In this thesis, we propose and evaluate Archipelago, our research platform to investigate and better understand object-oriented interaction in distributed applications. Archipelago is based on the concept of shared object spaces between parts of a distributed application, which contain active objects independent of individual parts. Through developing interaction technologies in Archipelago's shared spaces, they can be easily reused, extended, customized, and combined by means of the object-oriented principles of the application's own programming language. Furthermore, Archipelago's design allows to conveniently share application-specific data and computation.

We describe Archipelago's design and our prototype implementation, which replicates shared object spaces by adapting the replication technology of the Croquet project for our use case, and evaluate the platform by implementing illustrative examples.

Zusammenfassung

Verteilte Anwendungen bestehen aus verschiedenen Komponenten, welche miteinander über Verteilungsgrenzen hinweg iteragieren, um ein gemeinsames Ziel zu erreichen. Verschiedene Interaktionen benötigen unterschiedliche Mechanismen zur Kommunikation, Koordination oder zum Teilen von Daten oder Berechnungen. Folglich werden Technologien, die für diese Interaktionen eingesetzt werden, oft erweitert, angepasst und miteinander kombiniert, um applikationsspezifische Anforderungen zu erfüllen.

In dieser Arbeit präsentieren und evaluieren wir Archipelago, unsere Plattform zur Erforschung von objektorientierter Interaktion in verteilten Anwendungen. Archipelago basiert auf dem Konzept von geteilten Objekträumen zwischen Komponenten einer verteilter Anwendung. Diese Objekträume enthalten aktive Objekte, die unabhängig von einzelnen Anwendungsteilen sind. Interaktionstechnologien können in Archipelagos geteilten Objekträumen entwickelt und mit Hilfe der objektorientierten Prinzipien der anwendungseigenen Programmiersprache leicht wiederverwendet, erweitert, angepasst und kombiniert werden. Zudem erlaubt Archipelagos Design, anwendungsspezifische Daten und Berechnungen zwischen Komponenten zu teilen.

Wir beschreiben Archipelagos Design und die Implementierung unseres Prototyps, welcher geteilte Objekträume mit Hilfe angepasster Technologien des Croquet-Projekts repliziert, und evaluieren die Plattform anhand illustrativer Beispiele.

Acknowledgments

This work would not have been possible without the support and help from many people. I am deeply grateful to Prof. Robert Hirschfeld for his invaluable advice and guidance throughout the past year. I would also like to thank my colleagues for many insightful discussions and continuous encouragement, in particular, Patrick Rein, Tobias Pape, Johannes Henning, Toni Mattis, Tim Felgentreff, Marcel Taeumel, Andreas Raab, Lukas Schulze, Maria Graber, Philipp Tessenow, and Stephanie Platz.

Furthermore, I am deeply indebted to my fiancée, Alex, for countless morale-boostings, continuous inspiration, and never-ending tea supply and to my parents and family for providing relentless support, advice, and encouragement throughout the years. Finally, I would like to thank the Hasso Plattner Institute for supporting my studies with a generous scholarship.

Contents

1 Introduction **1**

2 Component Interaction in Distributed Applications **5**
2.1 Communication Models 6
2.1.1 Message Passing 7
2.1.2 Remote Procedure Calls 8
2.1.3 Other Models 9
2.2 Coordination Models 10
2.2.1 Publish-Subscribe 11
2.2.2 Message Queuing 12
2.2.3 Tuple Spaces 14
2.2.4 Dynamic Service Discovery 16
2.3 Sharing and Persisting Data 17
2.3.1 Shared Data Structures 18
2.3.2 Databases 19
2.4 Extension and Combination of Technologies 19
2.4.1 Extending Technologies 20
2.4.2 Combining Technologies 22
2.5 Summary 25

3 Coordination and Data Sharing with Archipelago **27**
3.1 Introduction to Archipelago 28
3.2 Concepts and Architecture of the Platform Design 29
3.3 Implementation of the Platform Design 32
3.3.1 Responsibilities of a Platform Implementation 33
3.3.2 Implementation Architecture Considerations 34
3.3.3 Other Implementation Considerations 37
3.4 Implementing Coordination Models and Data Structures . . . 40
3.5 Moving Application-Specific Functionality into Shared Spaces 43
3.6 Discussion 45
3.6.1 Limitations 46

3.7 Summary 47

4 Implementation of the Archipelago Prototype 49
4.1 Implementation Background 50
4.1.1 Squeak Smalltalk 50
4.1.2 The Croquet Collaboration System 50
4.2 Overview and Architecture of the Implementation 51
4.2.1 Replicated Object Spaces 52
4.2.2 Configuring the Distribution Boundary 58
4.3 Implementation of Croquet's Replication Technologies 58
4.3.1 Islands: Separate Image Segments 59
4.3.2 TeaTime: Computational Replication for Islands 62
4.4 Changes to Islands and TeaTime for Archipelago/S 67
4.4.1 Islands and TeaTime in Archipelago/S 69
4.4.2 Encapsulating the Distribution Boundary 74
4.5 Discussion 79
4.5.1 Open Questions 80
4.5.2 Limitations of the Implementation 82
4.6 Summary 84

5 Evaluation 85
5.1 ArchSpaces: Tuple Spaces in Archipelago/S 85
5.1.1 Tuple Space Interface and Implementation 85
5.1.2 Deploying the Tuple Space to Archipelago 88
5.1.3 Discussion of ArchSpaces 91
5.2 SpaceFight: An Example Application 95
5.2.1 Introduction to SpaceFight 96
5.2.2 Architecture of the Game 99
5.2.3 An Application-Specific Tuple Space 99
5.2.4 Application-Specific Functionality in the Island 101
5.2.5 Discussion of SpaceFight 102
5.3 Discussion of our Observations 108
5.3.1 Comparison with Related Approaches 111
5.3.2 Future Work 116
5.4 Summary 117

6 Conclusion 119

List of Figures

2.1 Asynchronous communication with sockets. 7
2.2 Synchronous communication with remote procedure calls. . 9
2.3 The publish-subscribe pattern. 12
2.4 The message queuing model. 13
2.5 Interaction with a tuple space for a master-worker scenario. . 15
2.6 Dynamic service discovery in a service-oriented architecture. 17
2.7 Architecture of the stock trading example. 24

3.1 Shared object space in the runtime systems of two participants 31
3.2 Two separate shared spaces within a distributed application 32
3.3 Application-specific functionality inside a shared space . . . 33
3.4 Archipelago implementation architecture choices 35
3.5 Moving the boundary between local and shared functionality 44

4.1 TeaTime replication architecture 53
4.2 TeaTime message replication 55
4.3 Boundary objects and auto message makers 74
4.4 Direct vs. indirect interaction with local objects 81

5.1 Class model of ArchSpaces 86
5.2 Extension of ArchSpaces with replicated services and jobs . . 92
5.3 Screenshot of the SpaceFight game 97
5.4 SpaceFight game architecture 98

List of Listings

4.1 Interaction with an object residing in a different local island . 60
4.2 Interaction with an object residing in a replicated island . . . 64
4.3 Setting up a Croquet world with the CroquetHarness. 68
4.4 Using the ArchipelagoHarness to create, join, and leave islands. 70
4.5 Example for labelling of local state and functionality in a boundary object. 75
4.6 Interaction with a shared object via auto message maker . . . 78

5.1 Interaction with an ArchSpaces tuple space. 87
5.2 Examples for labelling of TupleSpace operations 89
5.3 Extension of ArchSpaces with tuple depedencies 93
5.4 Extending the TupleSpace with multiple tuple lists 100
5.5 Excerpt of an application-specific interface to the TupleSpace 102
5.6 Excerpt of the StatisticsService implementation 103
5.7 Decoupling of the game logic and collision services via tuple space events 107

1 Introduction

Parts of distributed applications need to interact, that is, communicate and share or exchange data with each other, to coordinate their execution. Modern distributed applications make use of different communication and coordination models, including message passing, remote procedure calls, the publish-subscribe pattern, and message queues, as well as different kinds of shared data structures or databases to manage data shared by different parts of the application. Some models, such as tuple spaces, integrate communication, coordination, and data sharing in a single abstraction.

The same models and technologies are not always used for the same purpose and, because of this, different applications impose different requirements on the technology implementations. Many different general-purpose and domain-specific feature extensions to the original communication and coordination models have been proposed or implemented in order to increase the separation of concerns between the application and the technology and code and feature reuse between different applications, or to transfer the model to otherwise unsuitable application domains.

However, existing implementations of the models may only provide subsets of these features and it is typically difficult for the application developer to extend or customize these implementations to fit their specific application requirements, for example, to add a garbage collection feature or support for custom domain-specific template matching strategies to an existing tuple space implementation. Reasons for this include that the technology implementations are usually not aimed towards enabling custom extensions, but rather intended as a black-box framework, and that the developer is often unfamiliar with the particular dependencies and platforms underlying the implementations. As a consequence, developers often choose to put up with the unmodified general-purpose implementations, and application design suffers from restrictions put into place by inflexible technologies.

Furthermore, as no single communication, coordination, or data sharing technology is adequate for every task, a single application often makes

use of a number of different ones. A distributed stock trading application, for example, could use remote procedure calls for the communication between user-interfacing components running on client computers and a central database server providing them with quotation histories, a publish-subscribe technology to notify the user-interface clients of updates in stock quotations and a message-queuing system for the processing of trading orders. Similarly, a multi-player networked game may use a tuple space for general communication and coordination tasks between participants and a shared tree data structure to store sorted game data, such as statistics.

Because implementations of such technologies are hard to extend, application developers often choose separate technologies providing highly specialized implementations of the individual models over a technology integrating multiple less specialized models. When combining these separate implementations of different technologies for use in a single application, application developers may face the problem of integrating different middleware platforms, as each technology comes with its own platform dependencies. This can make the software more complex and, in turn, harder to maintain. It also often incurs higher configuration and deployment efforts.

In this thesis, we set out to evaluate an alternative approach to implement, deploy, reuse, and extend coordination and data structures within a single platform for distributed communication, which we call Archipelago. The Archipelago platform design is targeted towards object-oriented languages and allows the development and deployment of coordination models and shared data structures within the distributed application's runtime system and with the same programming language.

We achieve this by transparently sharing a part of the application's runtime system between distributed parts of the application. This shared space can contain multiple objects and object structures that can be accessed by different distributed parts of the application. Shared objects are mutable, active, and independent of individual participants. They can be interacted with by means of the programming language's message passing mechanism. Our prototype implementation of the Archipelago platform design is based on replicated shared object spaces and, for this purpose, employs Croquet's Islands and TeaTime technologies [45], which keep object spaces consistent by replicating object computation instead of object state, and extends them to fit our requirements. It also allows the convenient encapsulation of the distribution boundary and its configuration within the implementation of shared objects, making them easy to reuse.

Our platform allows the implementation of coordination and data structures within the application runtime, thereby enabling application developers to make use of the extensibility features of the application's own object-oriented programming language for extending and combining these structures to their specific needs. On top of this, developers can easily implement custom structures or give the responsibility for certain application-specific functionality into the shared space. The main limitation of our approach is its restriction to a single programming language, while our prototype implementation raises additional issues concerning various aspects of the interaction between local and shared objects, such as preserving the consistency of shared object state and choosing the most suitable parameter passing policies for messages sent to shared objects.

We expect the design of our platform to enable application developers to easily extend, customize, and refine reusable implementations of coordination models or shared data structures with missing features or domain-specific features and functionality and to allow the use of a single platform for multiple different coordination purposes. In consequence, we expect this to reduce application complexity and promote the separation of concerns within the application's implementation and ultimately enable a more modular application design.

Contributions

The intent of this work is to present and evaluate our ideas for the Archipelago platform. Our key contributions are as follows.

- The implementation-independent design of the Archipelago platform is based on a shared object space between parts of a distributed application. Shared objects are active, fully mutable, and independent of individual participants. Coordination models and shared data structures can be defined as object structures, which can be developed like local objects and later deployed to the shared space. Through applying object-oriented concepts for the development of shared objects, such as encapsulation, inheritance, and composition, shared structures can be easily reused and extended. Furthermore, Archipelago enables distributed applications to give the responsibility for part of their functionality to the platform by deploying it into the shared space, which can simplify their architecture.

- Our prototype implementation of the Archipelago platform is based on the transfer of Croquet's Islands and TeaTime technologies for the development of distributed applications with computationally different parts. We show how these technologies can be used to realize Archipelago's concepts. In particular, we focus on how to provide a convenient way for the interaction between shared and local objects that allows the simple reuse of shared object structures, such as general-purpose coordination models or shared data structures. Therefore, we propose the use of boundary objects and special proxy objects to encapsulate the distribution boundary within shared objects and allow them to interact with local objects.
- To evaluate our approach and the Archipelago platform, we apply our platform prototype to implement a reusable, extensible tuple space called ArchSpaces and an exemplary distributed application, the multi-player networked game SpaceFight, which extends ArchSpaces with application-specific functionality and combines it with a custom shared data structure.

Thesis Outline

In Chapter 2, we first provide an overview of typical communication, coordination and database technologies used in distributed applications. Chapter 3 then introduces and discusses the implementation-independent ideas and concepts behind the Archipelago platform design and explains how the platform can be used to define reusable and extensible coordination models and shared data structures. Subsequently, Chapter 4 presents and discusses our prototype implementation of the Archipelago platform within Squeak/Smalltalk, which we call Archipelago/S. It also discusses the Islands and TeaTime technologies of the Croquet project, which we reused and adapted for our prototype.

To evaluate our platform, we then describe and discuss how Archipelago/S can be used to implement ArchSpaces, a general-purpose tuple space, and the multi-player networked game SpaceFight in Chapter 5. SpaceFight employs Archipelago's concepts to adapt and extend ArchSpaces to better suit its use-case, to combine it with a custom tree data structure, and to give the responsibility for shared application logic into the platform. Finally, Chapter 6 summarizes and concludes this thesis.

2 Component Interaction in Distributed Applications

According to Tanenbaum [49], a distributed system is a collection of independent computers that appears to its users as a single coherent system. Consequently, a distributed application deployed within a distributed system is an application whose parts are executed on multiple computers. In modern distributed systems, the computers that make up distributed systems are often virtualized within larger parallel computers. Because of this reason, when talking about distributed systems in this thesis, we choose to refer to a collection of computation nodes connected by a network, instead of physical computers.

In order to coordinate their execution, the distributed parts (components) of distributed applications have to communicate, that is, exchange data, with each other. All of the communication technologies available for this task are ultimately based on the protocols supported by the network that connects the single nodes in the distributed system. Nowadays, the networks used are almost entirely packet-switching networks and, therefore, the communication within distributed systems is, on a low level, based on sending packets (or rather, messages) between the nodes.

As not all communication is best expressed through the mechanism of sending and receiving messages, other communication technologies try to abstract from this mechanism and provide application developers with a different model for communication that hides the underlying messages, such as procedure calls on remote services.

Furthermore, many tasks in a distributed application require some form of coordination, for example, locating service or resource providers, distributing work amongst several nodes or notifying potentially unknown dependent participants of events. In order to separate the communication needed to coordinate these tasks from the actual computation tasks of the application, different coordination models have been devised [37]. These models are designed to ease the coordination involving many nodes, often

hiding the individual communication participants from each other, for example, by establishing roles and relationships between them [1].

In addition to communication and coordination technologies, distributed applications also make use of shared data structures and databases to store shared, structured, searchable, or persistent data.

Often, several communication, coordination, and data sharing technologies are combined for the development of a distributed application, as they are each suited well for different kinds of tasks. We also argue that the extension of these technologies with domain-specific features can help improve the software architecture of such applications. However, both combination and extension of these technologies can also create problems, such as increasing application complexity or difficulties in development.

In this chapter, we first describe the major communication and coordination models (Sections 2.1 and 2.2), as well as shared data structures and databases (Section 2.3), and their respective use cases in distributed application development. Subsequently, we discuss how different technologies can be extended and combined in the context of a particular distributed application and analyze the related problems (Section 2.4). Finally, we conclude the chapter with a summary (Section 2.5).

2.1 Communication Models

The communication models and technologies outlined in this section provide mechanisms for the communication between two participants and form the basis for the coordination models and patterns described in the following section.

Communication between two participants can generally be categorized as either synchronous or asynchronous communication. When a participant communicates with its communication partner in a synchronous way, it waits for the partner's response to the latest message it sent to the partner, before continuing its own execution and sending any further messages. Analogously, when communicating in an asynchronous way, it does not wait for its partner's answer before returning to its own execution. We first describe the message passing model of asynchronous communication (Section 2.1.1) and then the RPC model of synchronous communication (Section 2.1.2) and briefly outline other models (Section 2.1.3).

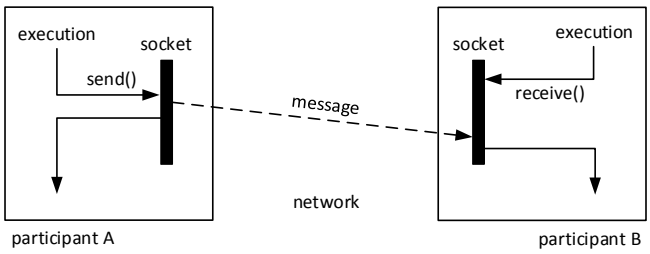

Figure 2.1: Asynchronous communication with sockets.

2.1.1 Message Passing

The message passing model is a direct mapping of the packet-based network protocols and allows a participant to send messages to and receive messages from other participants. A distinction can be made between transient and persistent message passing: Transient messages can only be received by participants that are available at the same time, while persistent messages can be sent while the receiver is busy or unavailable and are buffered by the communication technology and delivered when it becomes available. Typically, participants specify the recipients of a message explicitly, but implementations of this model can also allow broadcast or multicast messages to all or fixed groups of participants. The content of the messages exchanged can range from simple instructions to complex data structures or objects that are usually serialized for transfer, for example, in the form of XML [9] or JSON [13] data strings or protocol buffers [23].

One of the widely known implementation models of message passing is the Berkeley/POSIX sockets interface [50] shown in Figure 2.1. In this model, each end point of the communication is a socket that can connect and send messages to other sockets by specifying their network address and listen for messages sent to it, however, only transient messages are supported. A generalization of this model is the Message Passing Interface (MPI) standard [33], which allows participants to be identified by groups and identifiers instead of their network addresses.

The actor model [2] is a formalization of the message passing communication model with the communicating entities constituting concurrently executed actors that can reside on different nodes in a distributed system. An actor is an encapsulated process that cannot share any (global) state with another actor, but can communicate with it through message passing. Messages received by an actor are queued and processed by it one-by-one.

As response to a message, the actor can send messages to other actors, create new actors, and change the behaviour for the processing of the next message it receives.

The simple concepts behind message passing are the base for many other communication and coordination models, including remote procedure calls, message queuing and the publish-subscribe pattern. Message passing itself allows a direct communication between two or, in the case of group communication, more participants. This means, the participants have to know each other in order to communicate and implement custom mechanisms to solve coordination tasks. In a scenario where jobs have to be distributed amongst several worker nodes, for example, the application itself has to determine which node to send a job to.

2.1.2 Remote Procedure Calls

A remote procedure call (RPC) hides the message communication between the communication participants behind the interface of a procedure in the application's programming language. In the RPC communication model, two communication partners take the role of a client and a server, which provides a service used by the client. The idea of RPCs is to make calls to this service look and behave just like local procedure calls, both to the client and to the server.

RPCs are implemented by generating a piece of code for each procedure of the service both on the client and the server, which are called client stub and server stub respectively and encapsulate the communication between the client and server. As a client stub is a procedure with a signature equivalent to the procedure in the service interface, the client can simply call a client stub as it would call any other local procedure. The client stub then packages the arguments of the procedure call into a message and sends it via a message passing connection over the network to the server stub, which, in turn, unpacks the message and calls the procedure implementation of the service on the server. The return value of the procedure call is again packaged into a message by the server stub and sent back to the client stub, which unpacks and returns it as its own return value. This communication is shown in Figure 2.2.

Thus, typically, RPCs are of synchronous nature, that is, the client stub waits for a response from the server stub before returning the execution to its caller. However, there are extensions that allow an execution asyn-

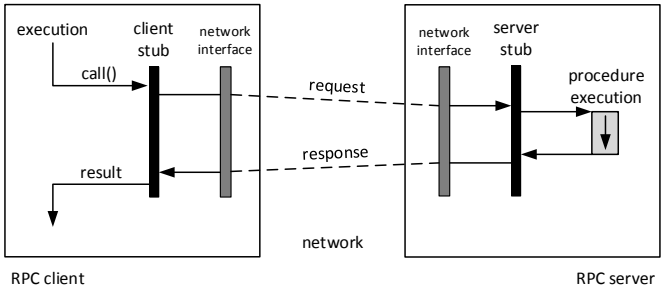

Figure 2.2: Synchronous communication with remote procedure calls.

chronous to the client execution and, for example, make the return values of remote calls act as placeholder until the response from the server has been received.

In the object-oriented world, the RPC equivalent is known as remote method invocation (RMI) and objects that communicate in such a way are referred to as distributed objects. Implementations of this paradigm include Java RMI [24], the Common Object Request Broker Architecture (CORBA) [21] and others.

RPCs are a popular solution to common client-server communication scenarios, for example between a user interface and business logic part of a distributed application. RPCs require at least the client participants to know the servers in order to obtain the service interface. However, many RPC platforms also incorporate service advertisement mechanisms that allow for dynamic service discovery by the clients, see Section 2.2.4. RPCs can also be used to simplify point-to-point communication between two participants from passing explicit messages to calling procedures, hiding the actual communication behind programming language procedures. The two participants then both play the role of client and server at the same time. However, as with message passing, RPCs are not a good fit for more complex coordination tasks, such as the job balancing scenario described above.

2.1.3 Other Models

Other models for communication between parts or processes of distributed applications are, for example, based on streaming data between them or sharing parts of their memory. Streaming technologies have become more important with the advent of multimedia applications, which often process

a continuous stream of data, and also tackle issues such as maintaining quality of service and synchronizing multiple streams with each other. Processes that share parts of their memory, on the other hand, can use this shared part both for communication and to exchange data. However, they often face the problem of synchronizing the access to data in this shared memory, which is why other technologies, including tuple spaces (Section 2.2.3), have been developed to extend the shared memory concept with further coordination mechanisms, and the shared memory model by itself is rarely used in the context of distributed applications.

2.2 Coordination Models

The communication models described above can already be used by themselves alone to manually solve coordination problems between parts of a distributed application. However, a number of coordination models and patterns have been developed to simplify some of the typical and reoccurring coordination tasks. Often, for these models, the aspect of loose coupling between the participants is a focus of interest.

Participants can both be coupled in space, by knowing their communication partners, and in time, by requiring their communication partner to be present at the time of communication. Distributed applications typically underlie changes in the set of their participants and their availability, for example, participants can dynamically be added or removed in order to allow for scalability of the application, and participants can become unavailable because of network or other resource failures. Therefore, both decoupling the participants in space, that is, participants no longer directly know their communication partners, and decoupling them in time, that is, participants can send messages to unavailable communication partners for later delivery, can be of interest.

Loose coupling between participants is advantageous not only for such dynamic changes, but also for software design purposes: Loosely coupled application parts enable a higher modularity and, therefore, are easier to extend, maintain, and evolve [34]. However, loose coupling between participants can also be disadvantageous: For example, decoupling in space can make it difficult to provide guarantees about the functionality of the application, as the participants cannot be sure which other participants

receive their messages or if any do at all, and decoupling in time can make it hard to provide transactional guarantees on the functionality.

In this section, we describe four different coordination models, namely, the publish-subscribe pattern (Section 2.2.1), message queuing (Section 2.2.2), tuple spaces (Section 2.2.3), and the dynamic service discovery pattern (Section 2.2.4), as well as their respective use cases and extensions.

2.2.1 Publish-Subscribe

The publish-subscribe pattern is a combination of the observer and mediator patterns known from object-oriented programming [18] with the purpose of enabling asynchronous, loosely coupled communication between parts of a distributed application. The pattern coordinates message (or event) exchange between multiple participants. Each participant can take the role of an event publisher or subscriber, or both. Publishers can generate events, which are then selectively passed on to subscribers by an intermediate participant, called mediator. The publishers, therefore, take the role of senders of messages in the communication, and the subscribers the role of receivers.

Consequently, a publisher does not know its subscribers, but only the mediator, and simply passes on all events generated to the mediator. A subscriber also does not know the event publishers directly, but instead registers its interest in certain classes of events at the mediator. When an event is sent to the mediator, it then notifies all those subscribers that had previously expressed interest in an event of the respective class; this process is depicted in Figure 2.3. The classification of an event can be based either on a topic associated with an event, such as a name or type, or its content.

The publish-subscribe pattern is used in cases where the sender of a message does or should not know its recipients and, likewise, the recipient does not care about the sender of the message. For example, in a stock trading application, there may be several different providers of stock quotations and many user-interfacing components that are interested in updates of specific quotations. In this case, the publish-subscribe pattern would be a perfect fit, allowing the users to express interest in certain quotation updates and the data providers to publish updates to a single central mediator.

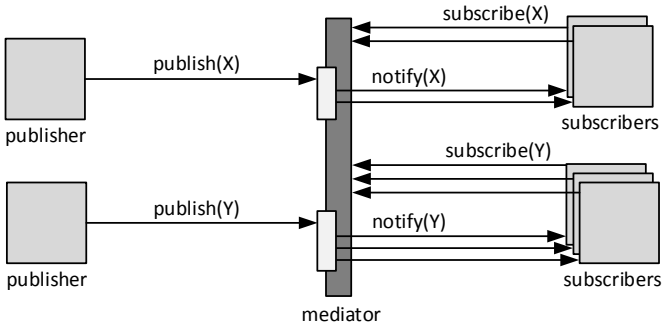

Figure 2.3: The publish-subscribe pattern.

Scenarios like this could also be solved by sending broadcast messages to all participants; however, the publish-subscribe pattern has two advantages: It takes away the burden of filtering the messages from the participants and it allows for a higher scalability, because the messages are only relayed to those participants interested in them. As discussed in the introduction of this section, the loose coupling between senders and receivers allows for dynamic additions or removals of the communication participants and promotes a more modular software design. However, this decoupling may also be disadvantageous: Depending on the implementation of the pattern, it can be impossible or difficult to guarantee that parts of the application function as expected, as producers generally cannot assume that specific subscribers are present and listening to the events they generate.

Technologies that implement the publish-subscribe pattern often provide additional extensions, such as authorization or security features or performance improvements. An example for the latter is to perform an additional filtering of events at the publisher, which can prevent publishers from generating events that no subscribers are registered for [43]. Additionally, several domain-specific extensions have been developed, for example, to allow the definition of special filters based on geometrical properties of shapes [29].

2.2.2 Message Queuing

Message queuing is a coordination mechanism that is closely related to the publish-subscribe pattern and is also based on passing messages between uncoupled participants via an intermediary. The intermediary manages

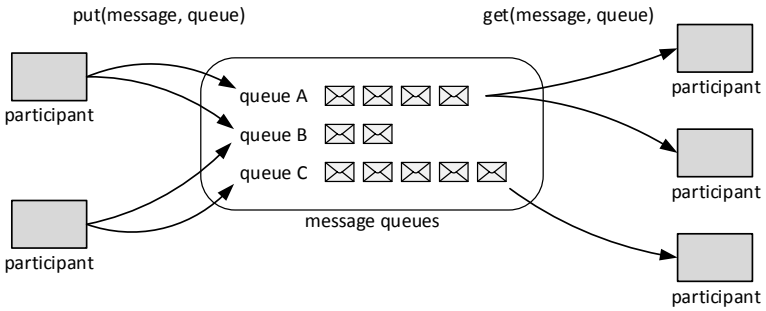

Figure 2.4: The message queuing model.

several named message queues, which participants can send messages to or retrieve messages from. Messages sent to a queue by one participant, the producer, are persisted until another participant, the consumer, actively retrieves them from the queue. Consequently, message queues not only decouple the participants from each other in space, but also in time.

The basic operations that a message queuing system provides the participants with are a put-operation that adds a message to a specific queue and a get-operation that can retrieve and remove a message from a queue (Figure 2.4). Most message queuing systems also allow an advanced configuration of the queues and their behaviour. The configuration of queues can, for example, allow messages submitted to the intermediary to be copied to several queues for different consumers to retrieve them, which can be used to implement a publish-subscribe-like behaviour. Even more advanced options can allow the redirection of new messages to different queues based on a prior classification of the messages by the intermediary [40, 16].

As with the publish-subscribe pattern, providing the participants with a way for asynchronous, decoupled communication between the participants is the main purpose of message queuing systems and, therefore, their goals, advantages, and disadvantages are similar. Comparing the two models, the intermediary in a message queuing system does not normally act as a notifying forwarder of messages as the mediator does in the publish-subscribe pattern, but rather as a temporary store for messages that participants actively retrieve messages from. However, extensions that add notification behaviour to message queuing systems also exist and, in such a case, a message queuing system can also be thought of as an extension to the publish-subscribe pattern that adds configurable re-routing of events to consumers and message persistence.

An exemplary use case of a message queuing system is a master-worker scenario, where one or more participants, called masters, publish jobs to be performed by one or more workers. Message queues are a natural fit for this scenario, particularly if the jobs do not have to return results to their master. The jobs can be added to a message (or job) queue and automatically distributed amongst the workers through the principle of retrieving messages from the queue. Of course, the advanced configuration mechanisms allow message queuing systems to be used for more complex coordination tasks as well.

Similarly to the publish-subscribe pattern, implementations of message queuing systems often also provide additional features - for example, to cancel messages sent to a queue [40], attach expiration times to messages [40], allow prioritization of messages [40], group messages into transactions [40, 17], or add authorization and security mechanisms [40, 17, 16].

2.2.3 Tuple Spaces

Tuple spaces have their origin in the Linda coordination language originally proposed by Gelernter et al. [19, 4, 11]. They are based on the shared memory model, formalizing it in a way that allows concurrent access to it without the need for additional synchronization of accesses and adding means of coordination. In Linda, all components of a distributed application, that is, the communication participants, access a shared tuple space, which they can add tuples to or retrieve them from.

A tuple in the sense of the Linda language is a typed one-dimensional data structure in the form of an ordered set of values. It is retrieved from the space through a template-matching mechanism: A template specifies the number of values in the tuple and, for each value in it, either the value itself, the type of the value or a wildcard. The tuple space provides four basic operations to add, read, and remove tuples from the space:

- `out` – to add a tuple to the space.
- `in` – to read and remove a tuple matching a given template from the space.
- `read` – to read, but not remove, a tuple matching a template from the space.
- `eval` – to emit a lambda-like process into the space that will eventually evaluate to a tuple, which is then added to the space.

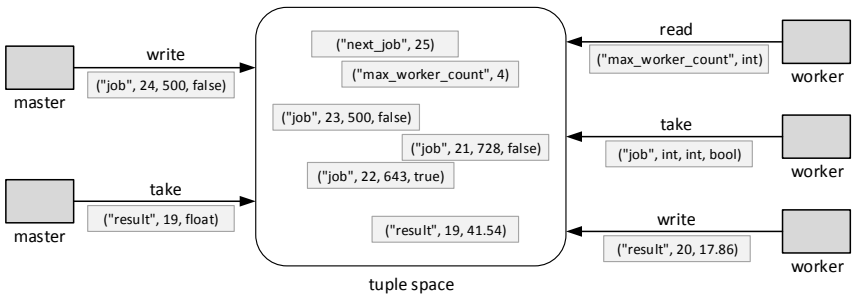

Figure 2.5: Interaction with a tuple space for a master-worker scenario.

The `in` and `read` operations are blocking: When a process queries the space for a non-existent tuple, the operation does only return control to the process when a matching tuple is added to the space. The two operations also do not give any guarantees on the order of the tuples returned by them.

The tuple space model allows parts of a distributed application to communicate with each other in an anonymous, decoupled, and asynchronous fashion, and access shared data in parallel, both by means of a single coordination abstraction. An example for using a tuple space as a means of coordination is, again, the master-worker scenario depicted before. The master can write tuples corresponding to job descriptions to the tuple space, and the workers can remove them from the space, as shown in Figure 2.5. The tuple space model also allows the workers to simply write back potential results of their work as tuples into the space, so that the master can then collect them. Tuple spaces can be used for other coordination tasks in similar ways [15], and even for publish-subscribe-like scenarios, where a message is distributed to many participants, if they are extended with operations to retrieve all tuples matching a specific template.

Just like message queues, tuple spaces aim to provide decoupling of participants both in space, by removing direct knowledge of each other, and in time, by persisting the tuples in the space. Additionally, tuple spaces are said to be able to reduce their control coupling, as the participants in the tuple space model do not send any actual messages to communicate with each other, and instead only produce and consume data tuples; however, the participants' knowledge about the kind of tuples produced or consumed incurs a higher data coupling. This can, for example, become

a problem during the evolution of an application if the format of certain kinds of tuples is changed.

Comparing tuple spaces with the other coordination models presented so far, the biggest difference is that tuple spaces not only provide means for communication between participants but also for storing shared data at the same time; in fact, they use shared data to communicate with each other.

Most of the implementations of tuple spaces for the major programming languages do not support the `eval` operation; however, almost all of them extend the basic Linda operations with further ones for reading and removing multiple tuples at once. Further extensions include notifications for added or removed tuples, configurable read/write policies and tuple matching strategies [14], transactions [31], and garbage collection [32], as well as domain-specific extensions, for example, support for storing location information [38] or other contextual information [39] with tuples and range-matching based on this information, or for semantic relationships between tuples [28].

2.2.4 Dynamic Service Discovery

Just as the previous three coordination models, the dynamic service discovery pattern has as its goal to decouple the communication participants from each other, but this pattern is geared towards client-server communication scenarios. It allows participants to act as a service provider in order to publish service descriptions at a central service registry and as a service consumer to search for a service by its description (Figure 2.6). This setup is also commonly referred to as a service-oriented architecture [8]. Often, the communication between service consumers and service providers is implemented via RPC mechanisms or extensions thereof, and the procedural interface of a service is used as its description.

A key difference between this pattern and the other coordination patterns is that the service discovery pattern only aids in the initial discovery of services and then allows the clients and servers to talk to each other directly, whereas the publish-subscribe, message queuing, and tuple space models employ an intermediary for all communication between participants. As a consequence, its use case is very narrow: it is highly specialized towards client-server architectures. To demonstrate the higher genericity of, for example, tuple spaces, consider that a simple service discovery pat-

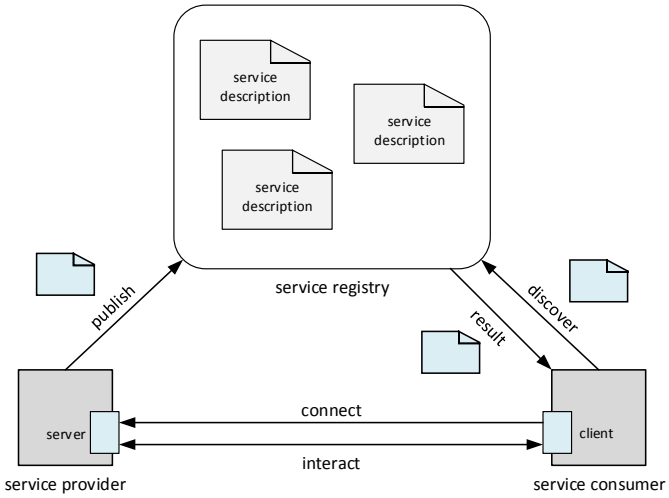

Figure 2.6: Dynamic service discovery in a service-oriented architecture.

tern could be implemented by a tuple space that acts as service registry; however, the opposite is not easily possible.

Many of the RPC frameworks also implement a service discovery mechanism corresponding to this pattern and some of them extend it with additional features such as leasing mechanisms to detect service or resource outages [53], support for mobile code [53], security and authentication mechanisms, or transactions [8].

2.3 Sharing and Persisting Data

Even though not all application architectures require it, parts of distributed applications also often share certain data between them in addition to communicating and coordinating their execution, enabling all parts to access or modify the shared data concurrently during their execution. In some cases, it can even be easier to use a shared data structure to solve a coordination problem, rather than using other communication or coordination models: Consider a simple distributed search for a maximum value in a large data set. It may be easier to share the access to a current maximum value between all participants during their execution than to use explicit communication models based on sending messages between the participants.

In this section, we describe two ways of sharing data between participants: by means of shared data structures (Section 2.3.1) and external database management systems (Section 2.3.2).

2.3.1 Shared Data Structures

The simplest form of a shared data structure is a shared virtual memory, as described in Section 2.1.3. It allows many participants concurrent access to simple structured data through low-level read and write operations. The participants will have to deploy a synchronization mechanism that prevents multiple participants from writing to the same location at the same time.

Tuple spaces (Section 2.2.3) are an extension to this mechanism with regard to sharing data between its participants. They provide synchronization through the Linda interface, but limit the structure of data shared to data tuples. A number of other common data structures have been implemented for use by multiple participants in the context of a distributed application, such as queue [51], graph [30, 20], tree [5, 36], and hash table [27] structures. We can distinguish between two general ways of implementing such structures: They can either be implemented as centralized services, or as decentralized structures that are distributed or replicated amongst the participants.

User-Defined Shared Data Structures

Different kinds of data structures are suited for different programming problems—some are even specialized to a specific problem [42, 3]. For example, certain kinds of tree structures allow range queries to be performed on partially ordered data, while graph structures can answer reachability questions on networks, and specialized trees, such as interval and segment trees, are used to determine intersections of geometrical shapes in space.

For this reason, some technologies apply more generic approaches to shared data structures which allow the application developers to implement and share custom data structures suited to the problem they are trying to solve. A simple solution is to implement the shared data structure as a centralized service that can be communicated with, for example, via RPC calls. Other technologies, such as the Orca programming language [6] and Virat platform [48, 47], provide support for distributed or

replicated structures. The Orca language, for example, lets the application developers specify abstract object data types and their operations. They can then interface with these objects as if they were local, while Orca's runtime system transparently assures their consistency and synchronization between the nodes.

2.3.2 Databases

Even more sophisticated systems to share data between parts of a distributed application are database management systems (DBMS). They are typically designed as black-box systems separate from the application, which allow data to be persisted across application restarts and provide the application with certain reliability guarantees, for example, towards the atomicity, consistency, isolation and durability of database transactions (the ACID property), and means to analyze the data stored in the database.

The common form of a database managed by a DBMS is known as relational database and stores data sets as rows in tables. It then allows different analytical operations based on the layout of the tables, for example, to combine or aggregate different data sets. Other forms of databases include object-oriented, document-oriented or graph databases.

The distinction between database management systems and shared data structures is not always easy: Some tuple space implementations, for example, have been extended with transactional concepts, persistence mechanisms and advanced querying mechanisms [31], which bring many concepts of DBMS to the tuple space model.

2.4 Extension and Combination of Technologies

In this section, we discuss how custom extensions to general-purpose technology implementations can help improve software design (Section 2.4.1), as well as why and how different technologies are combined in distributed applications (Section 2.4.2). We also analyze why extending technologies can be difficult and why combining different technologies can increase application complexity.

2.4.1 Extending Technologies

As described in the sections before, many technologies that implement the communication, coordination, or data sharing models extend these with additional features. We can divide these extensions into such that provide more advanced, but general-purpose, features and those that provide domain-specific ones.

For example, general-purpose extensions of a tuple space include support for transactions, garbage collection, and notifications. These extensions aim towards supplying the application developer with a better interface to the tuple space and pushing additional responsibilities that are common to many applications into the technology to further simplify application development. This can also benefit the separation of concerns between the application and the technology and promotes code and feature reuse between different applications.

Domain-specific extensions have a similar purpose, but are limited to a single domain of applications. Because of this, code and feature reuse is a less important goal; however, the separation of concerns is a driving force as well. For some domains, a custom extension of a model is even necessary to be able to support their use cases. An example for this is the extension of tuple spaces with support for range querying and data aggregation for use in context-aware applications [7, 35]: These features are necessary, for instance, to match on contextual location data overlapping geographic regions or to aggregate sensor values.

By promoting the separation of concerns and code reuse, such extensions are advantageous for the modularity and design of the applications making use of the technologies. But not all technologies implement the desired or required extensions, and even if multiple different technologies implementing the same model exist, they may not implement the desired feature set individually. In such cases, application developers have two choices: They can either choose to use an unmodified technology implementation and put up with the resulting restrictions, or implement the extensions or a custom technology themselves.

However, we argue that implementing custom extensions to an existing technology implementation can be difficult. Often, the technology is designed as a black-box framework and not aimed towards enabling custom extensions. Additionally, it may have its own platform and source code dependencies, which the application developers may not be familiar with.

In such a case, it may be hard for the developers to accomplish the task and they may lean towards using an unmodified technology implementation or implementing a custom technology instead. As an example case study to demonstrate these problems, we investigate how an application developer could extend the SQLSpaces tuple space implementation to support tuple dependencies in the following subsection.

Case Study: Extending SQLSpaces

The SQLSpaces tuple space implementation [55] uses a central server that hosts a relational database to store the tuples and runs a separate SQLSpaces server application written in Java. SQLSpaces further provides several client libraries that can be used to interact with the server from applications written in Java, Prolog, C#, and JavaScript. We assume a developer uses SQLSpaces for coordination within her distributed game application written in Prolog. In this game, multiple users can play a round-based card game. Therefore, a peer-to-peer game architecture is chosen: Each user runs a local Prolog program that connects to the tuple space. Each round, these Prolog programs create new tuples for the cards played in the round. In order to keep the development of her application simple, the developer would like to add a mechanism to the tuple space for automatically deleting these tuples after each round. For this purpose, she devises a new feature for the tuple space: Tuples should have dependent tuples, that are automatically deleted from the space when the owning tuple is deleted; therefore, the tuple space should support specifying the owning tuple when creating dependent tuples. Then, she can use this feature to make card tuples depend on the respective tuple describing the current round of the game, and when she removes the round tuple from the space to start the next round, all card tuples are automatically removed as well. In order to add this feature to the SQLSpaces implementation, the developer has to make several changes to multiple components of SQLSpaces.

We first look at the changes the developer has to make to the Prolog client library: First, in order to associate an owner tuple with each tuple, she needs to extend the representation of a tuple in the client library to allow storing, accessing and modifying this association. Next, she needs to make sure this association is also transmitted in requests to the SQLSpaces server. Messages between the SQLSpaces server and the client libraries are exchanged via a custom implementation of an RPC protocol based

on message passing. Because this protocol uses manual XML serialization for the tuples sent over the network, the developer also has to modify the corresponding serialization and deserialization code within the client library. With this, the changes to the client library are complete.

Next, we look at the changes she has to make to the SQLSpaces server Java application: As with the client application, the class representation of a tuple in the server library has to be modified to allow storing the association and the corresponding XML serialization and deserialization code also has to be updated. As tuples in SQLSpaces are stored in a relational database, she also needs to modify the Java/SQL code that dynamically creates the database tables for all tuples to add a field to these tables for the new association. Furthermore, the Java/SQL code that reads, inserts, deletes, or updates tuples in the database needs to be modified to comply with the new schema and read or modify the new field in the database. The developer now has to add the actual logic for removing dependent tuples when a tuple is removed from the space. Therefore, she needs to modify the code that is executed when the server receives a command to delete one or more tuple to first determine the dependent tuples (this involves defining a new SQL query) and then add them to the list of tuples to delete. With this last change, the changes to the server application are also complete.

In summary, the developer had to make a multitude of changes that involved the internal data representation, database storage, network communication, and command execution of the SQLSpaces server and the network communication and data representation in the client library. These changes not only required knowledge of the actual implementation, but also of the platforms and technologies underlying the server and client library: The developer had to be familiar not only with Prolog, but also with Java, SQL and XML. The design of the game, however, was improved by pushing the responsibility of keeping track of card tuples belonging to a round in the game into the coordination abstraction.

2.4.2 Combining Technologies

All of the technologies presented in the previous section have particular properties that make them well suited for different kinds of communication, coordination or data sharing tasks. Message queuing or tuple spaces, for example, are a good fit for a master-worker scenario, because the com-

munication has to be of indirect nature. However, in cases where the nature of the communication is a direct one, such as the communication between a client and a server, the decoupled nature of these two models may make both the communication harder than necessary and providing guarantees about the application's functionality more difficult, as discussed in Section 2.2. For such scenarios, RPC communication or actors would be better suited.

The different coordination mechanisms themselves also have different use cases: The publish-subscribe pattern is a perfect fit for event distribution, message queuing for master-worker scenarios and the service discovery pattern for dynamic client-server architectures. However, some of them also have overlapping use cases, for example, tuple spaces and message queues can both be used for the master-worker application architecture.

Similarly, different shared data structures and databases aim towards storing different kinds of data: A shared graph structure could, for example, be used for distributed path finding algorithms, whereas a tuple space or a relational database is less well suited for such a task. Likewise, graph structures would not be a good fit to store relational data, such as customer records.

Consequently, the decision whether a specific model, technology, or data structure should be deployed to solve a particular communication, coordination, or data sharing task depends on the individual use case. In large distributed applications, there may be more than a single use case for deploying such technologies. However, as we discussed before, different technologies may be suited better for different use cases respectively.

Consider the stock trading example from Section 2.2.1: For the communication between stock quotation providers and user interface clients, a publish-subscribe pattern for providing quotation updates fits well. To extend the example to allow users to look up quotation histories, we add a database storing those. The clients have to be able to access the data in this database, but for security reasons and to provide them with a simpler interface to access the data, the clients do not communicate with the database directly. Instead, they communicate with a central server via remote procedure calls to obtain the stock histories. We also allow users to participate in the stock exchange by buying and selling stocks through different stock markets. Therefore, we make use of a message queuing system that the clients can submit their orders to and which then routes

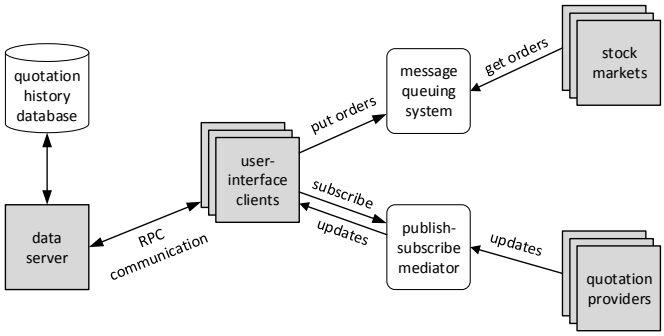

Figure 2.7: Architecture of the stock trading example.

the orders to the correct stock markets. The architecture of this extended example is shown in Figure 2.7.

In this example, the client component alone makes use of three different technologies: It acts as a subscriber in a publish-subscribe pattern, as a client in an RPC communication and as a message sender to a message queue. This combination of technologies is useful for the coordination between the individual participants in the distributed application; however, it can also have disadvantages: Each technology initially has to be set up and configured for use with the application. A technology implementing the message queuing model, for example, may require a node in the network that implements the role of the intermediary in the model, as well as the configuration of the redirection rules. This can make the deployment of the application more difficult. Additionally, the different technologies are usually accompanied by their own set of source code dependencies and may need initial setup and connection code within the implementation of the participants. Therefore, using multiple technologies can also make the application software more complex. In some cases, the requirements of different technologies may even be incompatible with each other, making it difficult to combine them without changes to their implementation.

When we compared message queues with the publish-subscribe pattern in Section 2.2.2, we noticed that message queues can also be used to implement a simple publish-subscribe pattern, when they are extended with support for notifications; the same is true for tuple spaces as well. In fact, many different coordination models can be used to implement simple versions of each other, and this provides for an alternative way of combining different coordination models. However, specialized implementations of

the individual models may provide additional features or domain-specific extensions and are often preferred, because the extensions of technologies that integrate multiple less-specialized models may be difficult, as discussed in the previous section.

2.5 Summary

In this chapter, we described a selection of existing models and technologies for communication, coordination, and data sharing in distributed applications: Message passing for direct asynchronous communication, remote procedure calls for direct synchronous communication, the publish-subscribe pattern, the message queuing model and tuple spaces for decoupled, asynchronous coordination, the service discovery pattern to decouple client-server architectures, as well as shared data structures and databases for sharing and persisting data.

We explained that these models and technologies have different use cases, advantages and disadvantages and that they are often extended with additional features. Finally, we showed that custom extensions to those technologies can improve the modularity and design of the application, but are difficult to implement, and that distributed applications often combine different technologies, which can have a negative impact on the application's complexity and deployment.

3 Coordination and Data Sharing with Archipelago

In Section 2.4, we have shown that extending the basic coordination models and data structures with advanced or domain-specific functionality is useful for software modularity and sometimes even necessary for the application of a model to a special use case, but that it is often difficult for the application programmer to extend existing implementations of these models with additional or custom functionality. We identified that the main reasons for this are that the technology implementations are typically designed as a black-box framework and that the developers may be unfamiliar to the underlying dependencies and platforms of these implementations.

We have also shown that even within a single distributed application, different coordination models and data structures are used. However, they are often not integrated into a single platform, but instead specialized implementations of different technologies providing advanced features are used, since it can be difficult to extend technologies that integrate multiple less-specialized models.

To solve these problems, we propose Archipelago, an alternative platform to implement coordination models and shared data structures. We introduce this platform in two steps: In this chapter, we focus on the implementation-independent design of the platform, and in the next chapter, we present a prototype implementation for Squeak/Smalltalk. This prototype is based on the Islands and TeaTime technologies of the Croquet project [45, 44] and many of the ideas behind the Archipelago platform were influenced by our experience with working with Croquet.

We first introduce the goals and ideas behind the Archipelago platform (Section 3.1), then describe its design concepts and the architecture of distributed applications built with it (Section 3.2). We further outline the responsibilities of an implementation of the platform design and the trade-offs of different implementation decisions (Section 3.3). Subsequently, we

describe how the platform can be used to implement, extend, and combine coordination models and data structures (Section 3.4) and to further move application-specific functionality into the platform (Section 3.5). Finally, we discuss the resulting advantages and limitations of the Archipelago platform (Section 3.6) and summarize the chapter (Section 3.7).

3.1 Introduction to Archipelago

The goal of the Archipelago platform design is to allow not only the easy use and reuse of coordination and data sharing technologies developed within it, but also the easy implementation and extension of these technologies at the same time. We argue that application developers should be enabled to customize the technologies, or even implement their own, with the development tools and platform that they are familiar with. For this reason, the Archipelago platform design is targeted towards object-oriented languages and allows the development and deployment of coordination models and shared data structures within the distributed application's runtime system and with the same programming language.

For the development of such models and structures, the developers should further be able to use the object-oriented concepts of the application's programming language, while the platform should abstract from the physical network communication between nodes and hide its implementation from the developers. Archipelago therefore allows the developers to create object structures that are shared between distributed parts of the application. They can design these structures as if they were local to a single site and then deploy them into a shared object space within the application's runtime by configuring distribution parameters or relying on platform defaults. This shared space is the main architectural construct of the Archipelago platform and can contain multiple objects and object structures that can be accessed and deployed by different distributed parts of the application, which we call participants of the shared space. It is the platform implementation's job to take care of the distribution and state synchronization of the structures within the shared space between the different parts of the application: The view on these structures should be consistent for all participants, and the structures should be independent of the individual participants. Furthermore, the platform implementation may provide additional guarantees, for example, concerning concurrency

synchronization of operations on shared structures or fault-tolerance of the platform against failures of individual participating nodes.

Additionally, in order for technology implementations on top of Archipelago to be easy to use within the application's own implementation, the distributed parts of the application should be able to interface with the shared object structures as if they were local ones. This way, the responsibility for the structure distribution is given to the developers of the structure and pushed into the platform. The object-oriented features of the application's language can further be used to hide the implementation of the structures from the application, giving the means for reusable and modular technology implementations.

We further realize that an implementation of these goals and ideas can have certain limitations, for example, not all programming language constructs may be supported for shared structures, and developers may have to make certain changes to the implementations of shared structures in order to deploy them to the shared object space within the runtime. Our intention is to keep these limitations to a minimum. However, we also believe that the developers of a shared structure should be provided with optional advanced tools that allow them to configure the distribution aspects of the structure, for example, in order to move the distribution boundary and share only parts of a structure's state and keep other parts local to the individual user of the structure. This, of course, makes these distribution aspects explicit and may create differences between the implementation of a shared and a local object; even so, it allows for complicated distribution scenarios and may be necessary for advanced implementations of shared structures or would otherwise restrict developers. However, for simple shared objects, an advanced configuration of these aspects should not be necessary and the platform's default behaviour should be sufficient. This way, we enable a transparent distribution in simple cases, and still allow advanced explicit distribution configuration for complex cases. We discuss the advanced configuration of the distribution boundary further in Section 3.3.3.

3.2 Concepts and Architecture of the Platform Design

As outlined in the previous section, the main concept of the Archipelago platform is a shared object space within the runtime system of the dis-

tributed application. This space can be shared by multiple participants, that is, distributed parts of the application, and contains objects in the sense of the programming language entity. Objects inside a space can interact and can be interacted with in the same way any other local object can, that is, by means of the message passing or method invocation mechanisms of the programming language. Further, they are fully mutable and can reference other objects inside and outside the space. They are active objects, that is, they can also exhibit behaviour in time. Additionally, they are independent of the individual participants: They do not require a particular participant to exist for the purpose of their own functionality. Consequently, if a participants deploys an object into the shared space, this object will then remain in the shared space even if the participant later leaves the space.

All participants share a consistent view on the objects within the space, that is, the return value of invoking a method on an object inside the space at a certain (logical) time should be the same, no matter which participant sent it, and its effects should be visible to all participants. Objects outside the space, however, are local to the participant's site, and only other local objects or objects within the shared space can reference them.

As a result of the shared spaces residing within the application runtime system, it is possible to implement the shared objects just like any other local ones. For a typical object-oriented language, this means that classes are defined for the shared objects, which have to be shared between all participants. As in object-oriented languages, these objects can reference other objects within the space, thus enabling the definition of complex object-oriented structures. Furthermore, the platform design allows for dynamic deployment of objects and object structures while the application is running. By providing support for naming of these objects, the platform allows participants of the shared space to look up objects by names or identifiers.

As mentioned above, the objects inside the space are active objects and can also exhibit behaviour in time, such as reoccurring tasks or actions executed when specific events are detected. This can be accommodated for by the platform implementation, for example, by attaching processes to objects or by providing the object with framework hooks for such tasks and actions.

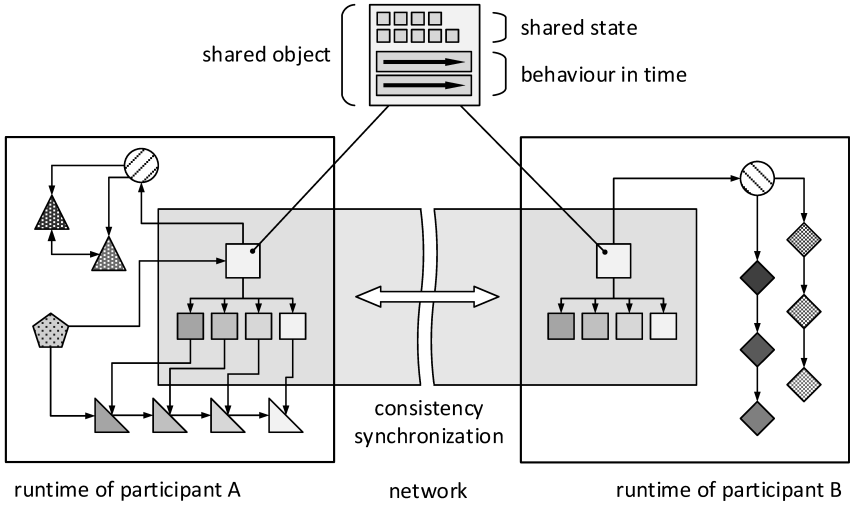

Figure 3.1: Shared object space within the runtime systems of two participants: The view on the objects within the shared space is consistent for both participants, however the objects outside the space are local to the respective participant.

These general concepts are depicted in Figure 3.1, which shows the architecture of the runtime systems of two participants connected to a shared space.

Complex shared object structures can be used to build data structures or coordination model implementations within the space, as we will show in Section 3.4. In large distributed applications, the architecture of the application may demand for different scopes of coordination or sharing data. For instance, in the example described in Section 2.4.2, the participants of the message queuing system (user clients and stock markets) and of the publish-subscribe pattern (user clients and quotation providers) are different ones. Therefore, it may make sense—both from a software as well as network architectural view—to divide the scope of the shared structures between these participants. For this reason, the Archipelago platform allows multiple shared spaces within the same runtime system that can have different participants. For the stock trading example, this means, that the application can use different shared spaces between the user clients and stock markets to implement the message queuing system, and between the user clients and quotation providers to implement the publish-subscribe pattern, as shown in Figure 3.2. This enables the separation of the objects

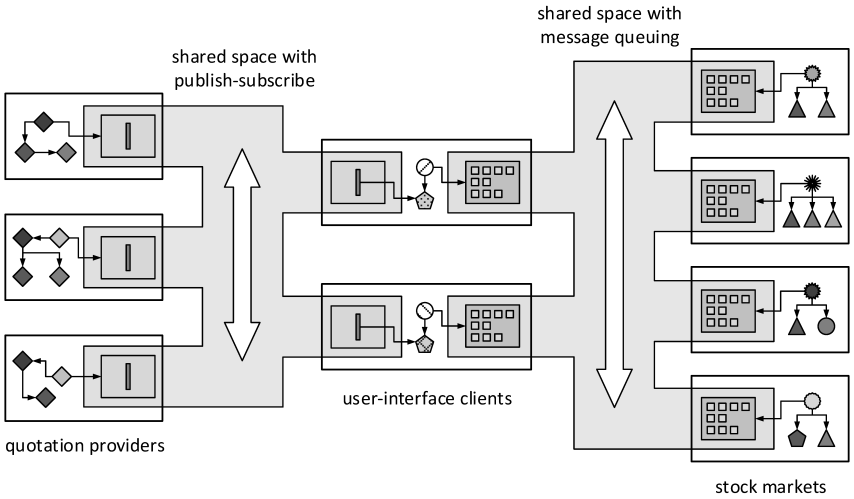

Figure 3.2: Two separate shared spaces within a distributed application. The space on the left implements a publish-subscribe pattern and the one on the right a message queuing system.

shared between the different participants. Note that, in order to keep up this separation, we do not allow objects within a shared space to refer to objects within another shared space.

The Archipelago concepts further allow the application developers to move parts of the application-specific functionality into the shared space, particularly custom shared application state and computation that operates on this state and whose effects should be visible to all participants. In the latter case, the developers can use this mechanism to push the responsibility of distributing and executing such computations into the platform. This can be particularly useful for peer-to-peer architectures where some central computation has to be performed on shared state, one example for this is the collision detection in a peer-to-peer game shown in Figure 3.3. We further discuss the prospects and consequences of moving application-specific functionality into the shared space in Section 3.5.

3.3 Implementation of the Platform Design

In order for the shared spaces to function as intended, an implementation of the Archipelago platform design has to fulfill certain requirements and responsibilities. In this section, we describe these responsibilities (Sec-

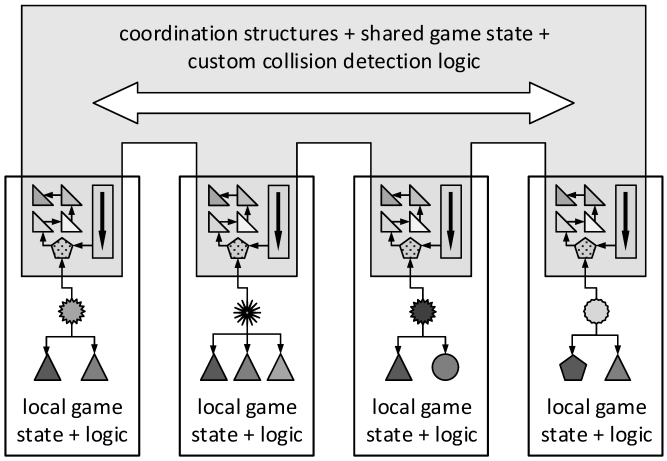

Figure 3.3: Application-specific functionality inside a shared space. Here, we show a distributed peer-to-peer game that uses the space not only for coordination, but also to store shared game state and perform central logic, such as collision detection.

tion 3.3.1) and discuss the trade-offs of different implementation architectures (Section 3.3.2) and a few other aspects that have to be considered when implementing the platform design (Section 3.3.3).

3.3.1 Responsibilities of a Platform Implementation

The most important and most complex responsibility of an implementation of the platform design is to maintain the consistency of objects within the shared space across multiple participants. The platform hides all necessary network communication, replication, and distribution implementation aspects from the developers and, therefore, these aspects are all responsibilities of the platform implementation. This means that the implementation has to transparently translate method invocations on shared objects to remote invocations where necessary, manage where shared objects reside and where their behaviour is executed, synchronize the state of potentially replicated copies of the same object and provide a way for shared objects to invoke methods on local objects outside of the shared space.

Furthermore, the platform should allow participants to dynamically join or leave a shared space. The platform implementation has to ensure that new participants obtain the same consistent view as all other existing ones,

and that participants leaving the space should not affect the objects inside the shared space, that is, even objects deployed to the space by the leaving participant should stay inside the space and have to continue to operate as intended. The implementation can provide additional reliability guarantees: Ideally, it should be tolerant to unexpected failures of individual participants and temporary network failures.

When objects are added to the shared space, other participants need to be able to find and access them. Therefore, the implementation has to provide a registration and naming mechanism to deploy objects or object structures to the space. It should be possible to give objects names when registering them, but, additionally, the platform implementation has to manage references to anonymous objects within the shared space that can be created and returned out of the space by other objects in the space. Therefore, the naming mechanism also has to support the dynamic naming of such objects.

In order to interact with the shared objects, all participants need to know the interface of these objects. We do not require that the implementation dynamically supplies these to participants during deployment of new objects, but also allow that all participants may have to be statically shipped with interface, class, and/or method definitions of all objects later added to the shared space.

Another aspect we allow platform implementations to take responsibility for is concurrency synchronization of the objects in the shared space. The implementation could, for example, automatically execute all operations on shared objects atomically. However, as the developers should be able to use all common language features within the shared space, we also allow that the platform leaves this task for the developers to solve with the programming language's tools for concurrency synchronization, such as locks and mutexes.

3.3.2 Implementation Architecture Considerations

As described in the previous section, the most complex implementation challenge concerns the consistency of the objects in the shared space across all participants. Three different implementation strategies come to mind, which can also partly be combined with each other: centralization, distribution, and replication of shared objects (Figure 3.4). In this subsection,

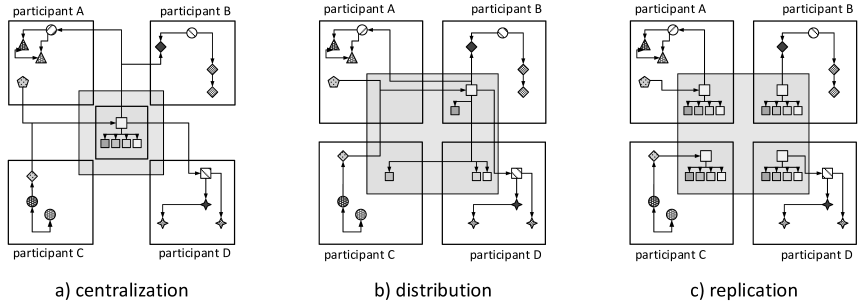

Figure 3.4: Archipelago implementation architecture choices: a) shows a centralized implementation, where the shared objects reside on a single separate node; b) shows a distributed implementation, where each object resides at one particular participant; c) shows a replicated implementation, where all objects reside at all participants.

we sketch implementations of each of these strategies and discuss their consequences.

Centralization

A centralized architecture of the shared spaces (Figure 3.4a) is probably the simplest approach to implementing the Archipelago platform concepts. In such an architecture, all shared objects would be transferred during deployment by any participant to a central server hidden in the implementation of the shared space. The platform implementation would automatically generate stubs for the objects in each participant's runtime when a shared object is looked up. These stubs can then communicate via remote invocations with the object residing on the central server. The behaviour of the shared objects is executed only within the runtime system of this server.

As objects within the shared space can also reference and interact with objects outside of it and local to a particular participant, the architecture also has to provide for the server to be able to invoke methods on the single participant's local objects. For this, a similar remote invocation approach can be used, which requires local objects to be dynamically named when passed to shared objects and proxied when accessed from within the shared space.

This approach is comparatively simply to implement, however, does not provide for scalability or reliability by itself: The single server can become a bottleneck for scalability and in case of its failure, the platform is no longer

accessible. However, centralization can be combined with replication for the central server to provide redundancy and increase reliability or with distribution of the shared objects onto several servers for scalability—but both replication and distribution are more difficult to implement.

Distribution

An alternative approach is to distribute the shared objects amongst all participants of the shared space (Figure 3.4b). In this case, every shared object resides within one participant's runtime system, for example, in the one of the participant that deployed the object to the space. Method invocations on a shared object are then either local operations if the shared object resides in the local runtime, or remote invocations if it resides in another one. The same is true for invocations on external objects from within the shared space, which also requires naming and proxying of runtime-local objects, just as in the centralized architecture presented before. In this distributed approach, the behaviour of shared objects would be executed by the runtime system of the participant the object resides in.

It would make sense to deploy distribution strategies that co-locate shared objects which interact frequently. For this reason, a platform implementation based on distribution of shared objects should also support relocating objects from one participant to another. This is particularly important for allowing participants to leave the shared space dynamically, while keeping the shared objects local to their respective runtime systems inside of the space. However, an issue with this kind of dynamic object migration is that participants generally do not know where the object resides physically. A participant cannot know whether an operation on an object will be fast, because it resides locally, or slow, because it resides on a remote site; therefore, it can only assume the worst case. The distribution approach by itself also does not support fault-tolerance: In order to additionally support the tolerance of participant failures, the approach has to be combined with replication mechanisms that keep redundant copies of all objects in the shared space.

Replication

The third alternative is the full replication of all objects inside a shared space across all its participants (Figure 3.4c). This allows for an easy access

to the shared objects, as each participant's runtime contains local copies of them; however, changes to their states have to be replicated and kept synchronized between different participants. This can either be implemented by method invocations that make changes to object state to all participants and synchronizing their execution, or by synchronizing object state in the background. Similarly, the behaviour of shared objects could either be executed at one (elected) location and the effects then replicated, or it could be executed at all participants in a synchronized way.

This approach is limited in scalability, as all changes of object state have to be replicated across all participants. However, it provides very good reliability and fault-tolerance towards failures of individual participants, as other participants can simply take over any potential responsibilities of failing participants. Replication can be combined with distribution, as described above, to achieve both reliability and scalability.

3.3.3 Other Implementation Considerations

Not only the way of maintaining consistency between participants is an important implementation aspect. Amongst others, further consideration should be given to the way in which parameters and return values are passed to or from method invocations on shared objects, how these method invocations are intercepted, where the distribution boundary resides, and how shared objects can interact with local ones.

Parameter Passing

When a method is invoked on a shared object, each of the argument values of this method call has to be sent to the shared object; likewise, the return value of the method invocation has to be sent back to the caller. We think this may happen in three different ways: First, a copy of the value can be sent; second, a remote reference to the value object can be passed; and third, the value object could migrate to a new location. The first approach is very simple: The value objects are copied and these copies are sent to the other side. After a method call, there will be a separate copy of each argument and of the return value on each side: a shared and a local one. If one of the copies is later modified, the other copy will not see any of these changes. For the second approach, references into the shared space and out of the space are created and passed instead; this means, there will

only be a single copy and modifications to this object from either side are visible to both sides. For the third approach, the location of the value object migrates from the location it is passed from to the location it is passed to: argument objects migrate into the space, and return values may migrate out of it. References to the migrating objects have to be updated to point to the new migrated object. As a result, objects may frequently change their location, but will generally follow the execution paths.

An implementation of our platform design has to consider which of these different parameter passing behaviours should be implemented for which cases. For example, one solution would be to adhere to the parameter passing standards of the underlying programming language, which is often based on the object types of the values. Most programming languages only pass the basic immutable data types by value, such as number values or strings, and all other objects by reference. However, if this policy is applied in a distributed objects setting, it could lead to long reference chains spanning over the network boundaries, which, in turn, can lead to performance problems. For example, it may not make sense to pass data holding objects, such as an object describing a 2-dimensional point, by reference instead of value. Further, this policy is not designed for objects that may also migrate from local to shared spaces. However, in our distributed setup, there may be cases where parameter value migration is the expected or desired behaviour. Therefore, the platform implementation may use different categorizations and policies for parameter passing and may even allow developers to specify the way that objects of specific types or in specific situations should be passed in.

Method Invocation Interception

Another aspect to consider is how references to shared objects are passed to objects outside the shared space: In some way, the platform implementation has to intercept method calls on these objects and replace or extend them with logic that implements the network communication for remote invocations and/or the synchronization of replicated copies. There are several different ways to accomplish this, such as proxying, dynamic subclassing, or dynamic stub generation; each requiring complex integration into the development process. Thought has to be given towards which of these mechanisms is suited best for the specific programming language and easy to implement. Some of the mechanisms may require advanced

meta-level language features, or even the extension of the language compiler.

Distribution Boundary

While we believe that shared objects should be interacted with and implemented by means of the same principles that local objects are, there are certain additional distribution aspects that the developers of a shared object should have control over. These aspects should be configurable during the deployment or within the implementation of a shared object, but the default configuration of these aspects should be sufficient for simple shared objects.

One such aspect is the location of the distribution boundary in the interaction between local and shared objects. In a centralized or distributed implementation architecture, this boundary is crossed by remote invocations of methods on the object, and in a replicated environment, it is crossed by synchronizing the effects of method executions across all copies. While, by default, this boundary should be crossed whenever interacted with a shared object, in special cases, it can make sense to execute some logic of the shared object before or without crossing the distribution boundary. An example for this could be the local caching of non-changing values of the shared object, so that later requests for these values will not require sending messages across the distribution boundary.

Therefore, we argue that the distribution boundary should be configurable by the developers of shared objects or object structures. Other researchers also state that crossing the distribution boundary should not be a completely transparent process, because developers have to deal with different problems, such as network latency and partial failure, when interacting with a shared object [54]. We argue that while it is true that crossing the distribution boundary has to be explicit in some cases, the main responsibility for this should lie with the developers of a shared object.

Therefore, it should be possible to hide crossing the boundary from users of the shared structures by encapsulating its configuration within their implementation or deployment. This allows the implementation of reusable shared object structures, such as coordination mechanisms or data structures, that encapsulate the distribution boundary from their users and can be used as if they were local. One way to solve this challenge would

be to allow developers to define a custom proxy object that handles the interaction with a shared object. Alternatively, the shared object's implementation itself may be able to specify which part of its functionality is to be executed before and which behind the distribution boundary.

Interaction with Objects Outside the Space

We allow shared objects to reference objects outside the shared space, so that they can, for example, implement advanced event handler registration functionality. This can, however, also make shared objects dependent on the existence of particular objects outside the space: A shared objects could hold a direct reference to a participant-local object and use this to implement parts of its functionality. This could make the shared object dependent on a particular participant, which Archipelago's concepts do not allow. Therefore, an implementation of the Archipelago platform design may have to impose additional constraints on the way in which shared objects can reference and use objects outside the shared space. Alternatively, the platform implementation could decide to support only an indirect mechanism to interact with objects outside the space. We discuss this issue in more detail in the context of our prototype implementation in Section 4.5.1.

3.4 Implementing Coordination Models and Data Structures with Archipelago

Coordination models and shared data structures can be implemented on top of the Archipelago platform as simple or more complex object structures in the shared space. In Figure 3.2, two coordination models deployed in two separate shared spaces are shown: The publish-subscribe pattern and the message queuing model. Both of these models are made out of a central intermediate that coordinates the communication between different participants by storing, categorizing or forwarding their messages. This is easy to implement in an object-oriented model: The intermediary becomes a first-class object in the shared space that the different participants can interact with. The intermediary can use additional private structures in the shared space for storing or buffering messages or similar purposes. In the publish-subscribe pattern, for example, it could hold a list of the

subscribers and provide a method interface for use by publishers to distribute events amongst subscribers. Other coordination models can be implemented in a similar fashion, as they all make use of some kind of intermediary that can be implemented as an object in the shared space. Shared data structures can by definition be transferred into an object structure within the shared space.

The developers of the model or data structure implementation can use the common object-oriented concepts that are also used for the implementation of ordinary local objects—in fact, they can implement the intermediary as if it were a local object and later apply the necessary distribution configuration and deploy it to the shared space. This means, they can make use of the advantages that object-orientation provides in regard to encapsulation, reusability, extensibility and combination of different models and data structures.

Encapsulation

First, object orientation allows the developers of shared object structure to hide the implementation of the structure from users of the structure. In object-oriented programming, this principle is called encapsulation and allows to change the implementation of an object without modifying its interface. It is designed to encourage the separation of concerns and enable reusability. Therefore, encapsulation is paramount for implementations of data structures or coordination models: Their objective is to provide users with an abstraction of their implementation, either to ease coordination or data organization.

Reusability

Enabled through the encapsulation of the object structure's implementation from its interface, the principle of reusability aims towards building object structures that can be applied in multiple contexts in one or more applications. This principle enables the development of coordination model or shared data structure implementations that can be reused across different applications or different use cases in the same application. For example, the publish-subscribe pattern implementation in the shared space from the example in Figure 3.2 can be built so that it can be reused for a similar purpose in a different application.

Extensibility

Object-oriented reuse is further supported by the principle of extensibility, which allows an object's implementation to be reused while refining parts of it, thereby enabling the extension of the object's functionality for a different or more specific purpose. This extensibility is supplied through the concepts of object composition and inheritance: With object composition, an object's functionality can be extended without revealing its implementation, and with inheritance, parts of its actual implementation can be extended or replaced.

An application developer can use these concepts to extend existing general-purpose implementations of coordination models or data structures with advanced or domain-specific features. We argue that this makes an implementation of such a model or structure on the Archipelago platform easier to extend than an external implementation, such as the SQLSpaces technology discussed in the example in Section 2.4.1. In this example, SQLSpaces was especially hard to extend because it was implemented in a language different to the application language, deployed a server program and a separate client library and used several additional technologies, such as an external database for tuple persistence and XML serialization for message exchange between the clients and server. A tuple space implemented on the Archipelago platform may still use a separate database to persist tuples, but no separate client and server implementations are needed, the tuple space is implemented in the same programming language as the application and message exchange is handled by the platform and hidden behind the ordinary message passing or method invocation semantics of this language. All these aspects can reduce the complexity of developing an extension to this tuple space implementation.

Combination

The concept of object composition further allows to combine multiple implementations of different structures and models within the same shared object space. As described in Section 3.2, the platform also allows to use different structures and models in different shared spaces within the same participant's runtime system. With the help of these two mechanisms, application developers can use the Archipelago platform to combine and deploy multiple different implementations, as required by complex appli-

cations. Because it is comparatively easy to extend these implementations with missing or domain-specific features, we further argue that we can lessen the advantages of deploying multiple specialized implementations of different coordination models or shared data structures on different platforms compared to the combination and extension of multiple less specialized ones on a single platform.

3.5 Moving Application-Specific Functionality into Shared Spaces

Besides enabling extensible implementations of coordination models and shared data structures, the Archipelago platform also allows to move application-specific functionality into the shared space. This functionality could be data structure or coordination logic specific to a single application—in this case, it would most likely be implemented as an extension to a shared data structure or coordination model, as described in the previous section. Additionally, for some applications, there may be functionality that does not belong to a single participant, for example, because it is functionality that no single participant should be responsible for or because the application developers do not really care which participant executes it. In such cases, the developers can implement this functionality as shared state and shared computation contained in shared objects and push the responsibility of its execution into the platform by deploying the objects into the shared space. The platform then has to make sure that the functionality is executed and that the effects of this execution will be visible to all participants. Advantages of this include that the platform can identify a location to execute the functionality that is well suited to do so (for example, from a performance view point) and that the execution is independent of any particular participant being available, that is, its reliability is guaranteed by the platform.

Figure 3.3 shows the example of a peer-to-peer multi-player networked game implemented on top of the Archipelago platform. In this case, every participant of the shared space has a similar role (that of a player) and, therefore, no particular participant should be responsible for logic that has a central aspect. For this game, such logic is the collision detection logic, which detects if two entities in the game are colliding and if so, executes the respective action associated with the collision. Therefore the

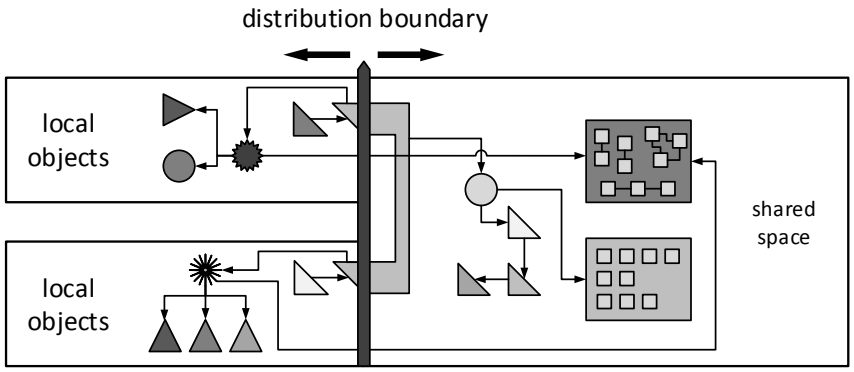

Figure 3.5: Moving the boundary between local and shared functionality. Here, we see two different participants on the left side and a shared space between the two on the right. The distribution boundary divides local from shared functionality and, in this case, also splits the functionality implemented by a single object.

collision logic is encapsulated in a shared object or object structure and deployed into the shared space, next to other shared data and coordination structures. This allows the execution of the collision logic independent of the connected participants, and additionally ensures that no single participant can manipulate this logic.

In Section 3.3.3, we discussed that the distribution boundary between local and shared objects should be configurable by the developers of a shared object, which could allow part of an object's functionality to be executed locally and other parts in the shared space. As shared objects are otherwise implemented just like local ones, we believe that this makes the boundary between local and shared application-specific functionality very flexible. As a result, the developers can easily move the distribution boundary between the local and shared functionality on intra- and inter-object basis, as depicted in Figure 3.5. For example, the developers could use this flexibility to move parts of the collision detection functionality to specific participants in order to allow them to handle collisions of entities owned by them in a special way.

3.6 Discussion

We argue that by enabling implementations of coordination models and shared data structures within the application's own runtime system and programming language, the Archipelago platform makes it easy for the application developers to extend and combine these implementations. This is made possible both by the object-oriented concepts applied for their development and by the developers' familiarity with the language.

Shared object structures as the means to implement shared data structures and coordination models allow to hide the network communication, distribution, and consistency responsibilities from the developers of these structures. Instead, the platform takes over these responsibilities and provides optional configuration mechanisms to adjust the distribution behaviour where necessary, such as fine-grained control over the distribution boundary. This makes it possible to implement shared structures just like local ones and later adapt and deploy them to the platform by adding them to a shared space in the runtime system of the application. It further allows to develop and deploy custom coordination or shared data structures that are targeted for a specific use case or to move parts of the application's functionality into the shared space. We argue that this can improve the separation of concerns in the application's implementation and, therefore, its modularity.

Furthermore, the users of shared object structures can interact with these structures as if they were local, which maintains encapsulation of the distributed nature of the shared structures and therefore enables reuse of these structures and, with that, of the shared data structures and coordination models implemented on top of the Archipelago platform.

We also argue that the Archipelago platform can be used to combine implementations of multiple different data structures or coordination models within the same platform as a result of these implementations being easily extensible. Because implementations on top of the Archipelago platform can be extended to the application's needs, the benefit of using a separate specialized technology instead, which does not require extensions, but requires configuration and deployment of a separate platform, is lowered significantly.

3.6.1 Limitations

The biggest limitation of the platform design as presented so far is that it is restricted to applications whose parts are all written in the same programming language. The reason for this is that we expect the implementation of a shared object to be transferable from one part of the application to any other part, in order to allow the platform to distribute it freely amongst the different parts. One way the platform design may be extended for support of multiple different languages could be automatic source code translation into different programming languages. However, in this case, coordination model or data structure implementations may be developed in a programming language different from those parts of the application that will use them, possibly making it harder for developers of these parts of the application to modify them to their needs. Therefore, we currently regard this limitation as a design limitation that cannot be solved simply without sacrificing the ease of object extensibility.

Furthermore, the optional configuration of distribution aspects, which developers of shared object structures can make use of, may violate the concept that shared structures should be implemented just like local ones. This means that in order to extend structures that require additional configuration to be deployed in the shared space, the developers have to have knowledge about the Archipelago platform implementation in so far as to understand this configuration. However, we believe that the scope of this required knowledge is limited and that the benefit of using the same object-oriented abstractions for the development of shared as well as local objects outweighs this limitation.

The platform design presented in this chapter is implementation-independent, but we have left many issues that need solving to the implementation of this design. These issues include the way in which participants of a shared space share the code of the shared objects and their classes, the way in which parameters are passed between local and shared objects, how the distribution boundary can be configured, and how shared objects can interact with local ones. The platform implementation needs to provide mechanisms that solve these issues, either in rather simple or sophisticated ways.

3.7 Summary

In this chapter, we introduced the design concepts of the Archipelago platform. Its purpose is to enable reusable and easily extensible implementations of coordination models and data structures structures within a single platform. Therefore, the platform employs the concept of a shared object space within the runtime system of a distributed application, which also allows developers to move parts of the distributed application's functionality into the shared space.

We further outlined the responsibilities and requirements of an implementation of the platform design. These primarily include maintaining a consistent view on the shared space from all its participants, support for dynamic changes in the set of participants and a naming mechanism for the shared objects. We then described and compared three different architectures of such a platform implementation—a centralized, distributed, and replicated one—and discussed other issues that it has to solve, for example, the parameter passing policy or the configuration of the distribution boundary.

Subsequently, we described how the Archipelago platform can be used to implement and extend coordination models and shared data structures and to move application-specific functionality into the shared space. Finally, we discussed the advantages and limitations of the platform design; the predominant limitation being its restriction to applications whose parts are all written in the same programming language.

4 Implementation of the Archipelago Prototype

Our prototype implementation of Archipelago is based on the Squeak/Smalltalk platform [25]; we refer to it as Archipelago/S. For this prototype, we chose to investigate a replicated approach to our platform design. We expect a replicated shared object space to provide advantages over a centralized or distributed one, particularly regarding availability and redundancy, as discussed in Section 3.3.2, and to open interesting possibilities for the configuration of the distribution boundary, for example, to allow non-modifying accesses to shared objects without the need for network communication.

Traditional approaches to replicating objects are based on replicating data: Whenever an operation changes the state of a replicated object, the new state is replicated to all copies. In order for this to work, the replication infrastructure has to detect changes to objects and incorporate a locking mechanism to prevent concurrent conflicting changes on the same object. Archipelago/S is based on replicating computation instead of data and, for this purpose, makes use of the Islands/TeaTime object replication technology from the Croquet system [45, 44]. Replicating computation instead of data can often be easier and more efficient, as, for example, no locking is necessary and multiple changes to an object's state can be expressed with a single computation [46].

We first provide some context by outlining the image concept of the Squeak/Smalltalk platform and the purpose and ideas of the Croquet collaboration system (Section 4.1). We then describe our implementation choices and the general architecture of Archipelago/S (Section 4.2). Afterwards, we explain the implementation of the Island and TeaTime technologies of the Croquet system and how shared objects can be interacted with by means of these technologies (Section 4.3). We then present our modifications and extensions that adapt these mechanisms for use in Archipelago/S

(Section 4.4). Finally, we discuss our prototype implementation (Section 4.5) and summarize the chapter (Section 4.6).

4.1 Implementation Background

Our prototype is implemented on top of the Squeak Smalltalk programming language and environment [25] and makes use of the Islands and TeaTime technologies of the Croquet collaboration system [45, 44]. In this section, we outline Squeak's concepts relevant to understanding our prototype implementation and the purpose and ideas of the Croquet system.

4.1.1 Squeak Smalltalk

Squeak [25] is a highly portable Smalltalk-80 implementation and, therefore, a dynamic object-oriented programming language that is well suited for the fast development of prototype systems. Squeak's virtual machine itself is written in Smalltalk, which makes it easy to debug, analyze, or change its implementation. Squeak is purely object-oriented, that is, everything is an object. Robust reflection and meta-level programming features allow the dynamic analysis and extension of programs during their runtime; this is useful for prototyping Archipelago, because dynamic proxying of messages, as well as modifications of the VM can be easily realized.

In Squeak, all source code (both Squeak's programming language environment sources and the application sources) as well as all objects making up the state of the application are stored inside a persistable object memory, which is referred to as an *image*. As a result of this, applications can be suspended and written to disk at any point in time and computation can be restarted where it left off by loading the image back into memory.

We also chose Squeak as basis for our prototype, because we intended to use Croquet's technology, and Croquet is implemented on top of Squeak. Computation in Squeak is bit-identical on all hardware platforms, which is an important requirement for Croquet's replication mechanism.

4.1.2 The Croquet Collaboration System

Croquet [45, 44] aims to provide a 3D real-time collaboration platform for multiple participating users. It can, for example, be used for collaborative data visualization, virtual learning, or 3D wikis.

Applications in Croquet are deployed into an object space fully replicated between all participants. This object space is referred to as an *island* and can contain shared, replicated objects, which can also exhibit behaviour. Croquet uses a replication mechanism called *TeaTime* [45], which is based on replicating computation rather than synchronizing state. That is, operations on a shared object are synchronized and executed in the same order at all participants. TeaTime was originally based on Reed's work on object synchronization [41] and intended to be a pure peer-to-peer architecture [45]; however, the current implementation of TeaTime in Croquet requires a central message router to perform the synchronization.

Because of the nature of applications in Croquet, the objects making up a Croquet application are usually all deployed into the shared space and the resulting application architecture consists of multiple identical participants in a peer-to-peer setup. The only parts of such an application that are not shared belong to the user-interface framework, such as rendering primitives or device events.

For use in Archipelago, we separated the Island and TeaTime technologies from the Croquet-specific use-case of fully replicated 3D collaboration applications, ported it to Squeak 4.5, and generalized it for use in distributed applications comprised of computationally different parts. Because of this different use case, the focus on the replication technology is different, too: While in Croquet, the emphasis is on the interaction of replicated objects amongst each other, for Archipelago, it is on the way in which local objects interact with shared ones.

4.2 Overview and Architecture of the Implementation

For Archipelago/S, we specialize the concepts of the Archipelago platform presented in Chapter 3 towards a replicated object space. The general properties also apply for Archipelago/S: Shared objects can be added to this replicated space, can be interacted with just like local objects, and are active. Furthermore, they are independent of individual participants and consistent amongst them.

As described in Section 3.3.1, the Archipelago platform design delegates certain responsibilities to its implementation. Archipelago/S uses the Islands and TeaTime technologies from the Croquet system to carry out most of these responsibilities. These technologies provide a replicated shared

object space, called *island*, and with this, consistency of shared objects between participants and transparency of network communication. They also allow participants to dynamically join or leave a shared object space and provide registration and naming of shared objects in the island. Further, they tolerate faults of individual participants, with limitations discussed in Section 4.5.2. Additionally, operations on shared objects within an island are concurrency synchronized.

In the context of Archipelago/S, we further investigated solutions for the configuration of the distribution boundary and parameter passing policies, as discussed in Section 3.3.3. Our prototype implementation makes it possible to encapsulate the distribution boundary of shared objects within these objects, by dividing a shared object's data and functionality into two parts: one that is local to each individual participant and another one that is shared and replicated amongst all participants. This takes advantage of the fact that shared objects are replicated and therefore, a local copy of a shared object exists at each participant.

Our current implementation only allows parameter passing by reference for arguments that are shared objects and passing by value for argument objects that are local to one participant. Return values from replicated messages are passed by value or reference: Objects of basic or immutable types are passed by value, objects of other types by reference.

However, alternative solutions to our current parameter passing solution and to our current solutions for interaction between local and shared objects are still open research questions, which we discuss in Section 4.5.1.

In this section, we first describe the implementation architecture of the replicated object spaces in Archipelago/S, which is based on the Island and TeaTime technologies, and then outline how the distribution boundary for the interaction with shared objects can be configured in Archipelago/S.

4.2.1 Replicated Object Spaces

Archipelago/S uses the TeaTime architecture [45] to keep the individual copies of a replicated shared object (*replicas*) consistent between all participants. As mentioned before, this is accomplished by replicating computation, that is, executing operations performed on a shared object on each replica. For this purpose, the order of operations has to be synchronized and all operations have to be deterministic.

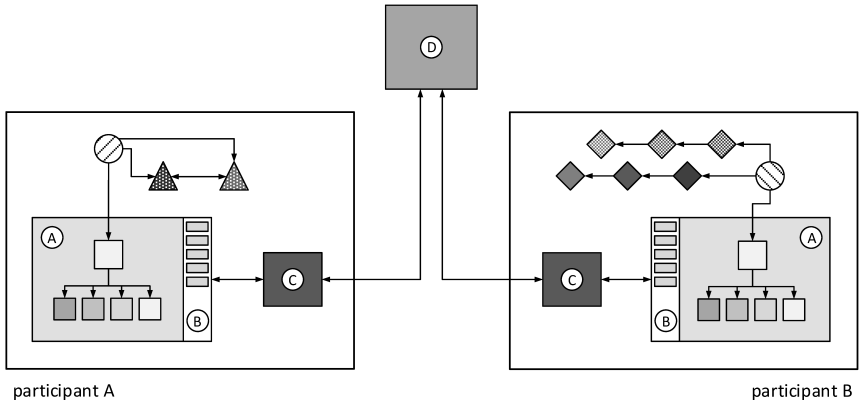

Figure 4.1: TeaTime replication architecture. The figure shows two replicas (A) of a replicated island and their island controllers (C) in two participants connected to a message router (D). Each island replica uses a local message queue (B) for executing replicated messages.

The TeaTime architecture depicted in Figure 4.1 implements this mechanism. All replicated objects are contained within a replicated object space called *island*. Each island replica is associated with a controller process, which connects to a central message router. Messages sent to an object inside the island are not actually sent to the object directly, but instead sent to the message router. This router serializes the message order by assigning each message a logical time stamp and relays these messages to all island replicas by sending them to the respective controllers. The controller in turn schedules these messages for execution by adding them to a sorted message queue, which is sequentially processed by the island.

Furthermore, TeaTime allows shared objects to exhibit behaviour by sending messages to themselves or each other that are scheduled to be executed at some point in the future. It also implements a mechanism that allows participants to join an existing island at any point in its execution. In the following, we describe some details of these mechanisms. Further information about Croquet's Islands and TeaTime architecture can be found elsewhere [46].

Replicating Messages to Shared Objects

Messages that are queued to be sent to objects inside an island are themselves data objects made up of five parts: the receiver object identifier, a

selector (message name), a variable number of arguments, a time at which to execute the message, and a message number. They are kept locally within each island replica in a message queue that sorts pending messages by the time they should be executed.

TeaTime distinguishes between two types of replicated messages: *internal* and *external* messages. Internal messages are messages from a shared object sent to another shared object or itself, which we discuss later. External messages are messages from objects outside the island. The process of sending an external message to a shared object is illustrated in Figure 4.2. When a local object sends a message to a shared one, this message is intercepted by a proxy object called *FarRef*; we discuss the implementation of these proxies in Section 4.3. The FarRef proxy wraps the message into a message object with receiver, selector, and arguments as described above and asks the local controller of the island replica to serialize and forward the message to the router.

The router provides the time base in TeaTime's architecture. When the router receives an external message from an island's controller, it timestamps the de-serialized message with the current time provided by its operating system. The router receives such messages sequentially, which ensures that no two external messages receive the same time stamp. After stamping the message, the router relays the message to all island replicas by serializing and sending it to the respective controllers.

When a controller receives a message relayed by the router in this way, it adds the message to the sorted message queue at the correct position based on the timestamp of the message. A separate process associated with the island replica continuously processes this message queue up to the latest external message in the queue. This will eventually lead to the execution of the received message: The message object is read and a corresponding Smalltalk message is sent to the receiver denoted in the message object.

By means of this mechanism, consistency of all side-effects of replicated messages is guaranteed as long as all operations performed on shared objects are deterministic, since messages sent to shared objects are executed at every participant in the same order. The network communication is hidden completely behind object-oriented message-based interaction: The TeaTime implementation takes care of serialization and network transfers. Furthermore, external messages are concurrency synchronized as well: As all external messages are executed by a single process, replicated operations are automatically executed atomically. This, of course, also means

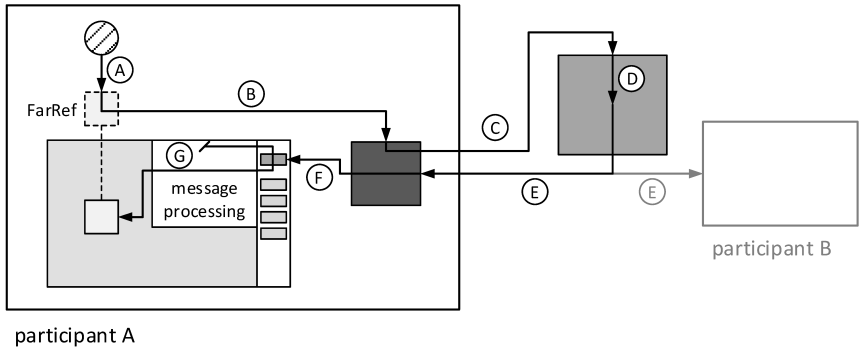

Figure 4.2: TeaTime message replication. The replicated is message is sent to the shared object through a FarRef (A), which forwards the message to the local island controller (B). The controller sends the message to the message router (C), which timestamps it (D) and distributes it to all controllers connected to the router (E). The controller then adds it to the message queue (F). From there, it is asynchronously removed by the island's message processing and sent to the shared object (G).

that the execution of messages inside a single island replica cannot make use of parallel execution to improve its performance.

Croquet's TeaTime implementation also allows return values of external messages to be sent back to the original external sender object via an asynchronous mechanism that we explain in Section 4.3. Archipelago/S further allows to hide this asynchronicity from users of shared objects, as discussed in Section 4.4.

Active Objects

Shared objects can exhibit behaviour in time by sending messages to themselves or other shared objects that are scheduled to be executed at some point in the future, that is, after a specific time delay. These messages are called internal messages and are also replicated at all participants.

When a shared object sends such an internal message during a replicated computation, it is directly added to the local sorted message queue of the island replica at the correct position depending on the delay specified. This delay is always calculated relative to the current *island time*, which does not change during the execution of a replicated message and is set to the timestamp of the message that is currently executed.

Internal messages do not need to be relayed through the message router, since they are added at the same point in the execution at every island replica. The island time ensures that each replica assigns the same time to a new internal message. However, because a shared object can send multiple internal messages with the same delay during the execution of the same replicated message, an additional message sequence number is assigned when internal messages are scheduled and ensures a deterministic order of two messages with the same timestamp in the message queue.

Internal messages may be scheduled for a time far in the future. This means, other external messages may arrive after the internal message was scheduled, even though they are scheduled to be executed at a time before the internal message's scheduled time. Because of this, islands only process messages in the message queue up to the latest external message timestamp. Internal messages queued for execution after this message remain in the queue until an external message with a later timestamp arrives. However, the progression of internal messages is assured even if no explicit external messages are sent: The message router periodically sends heartbeat messages to all island replicas, which contain the current time at the router. Island replicas can then execute the message queue up to this time.

Sharing Objects

In order to share a new object into the island, a participant can instruct the island to create a new instance of a given class, or copy an existing local object into the island. The participant is further able to specify a name for the shared object, under which other objects or participants can request a reference to it from the island. Alternatively, the island will provide a generated identifier. After adding a new object to the island, the island returns a reference to this object to its creator, which can be used to further interact with the object.

Because each participant needs to be able to perform the same operations on shared objects, it is imperative that all participants have access to and use the same version of the implementation of shared objects. This means, the same class versions have to be deployed to each participant's local Squeak VM. TeaTime does not provide for an automatic source code or class deployment, and leaves this responsibility to the application de-

veloper. For Archipelago/S, we currently follow the same strategy and support for automatic deployment is a topic for future work.

Joining and Leaving the Island

TeaTime allows participants to dynamically join or leave an island. When a new participant joins an existing island, it creates a local controller for its (to be created) island replica. This controller contacts the corresponding message router and requests new messages for the island to be sent to it. These messages are retained inside a buffer in the controller until the new island replica is set up. The controller then requests a copy of the island to be sent to it. Therefore, it sends a message to the router, which then relays this message to the individual controllers of other island replicas. One of these controllers will offer to provide the initial island copy for the new participant. This controller then snapshots its current local state of the island including the message queue and list of exported objects of the island and the state of all objects inside the island. For this purpose, the controller makes use of the fact that an island is implemented as a separate image segment in Squeak, as discussed in Section 4.3. This way, the island state can be snapshot without a long interruption of message processing. Once the snapshot is created, it is sent to the controller of the new participant, which then installs the snapshot into its local runtime system. Execution is continued by merging the message list buffered in the new controller with the message queue inside the installed island replica: Messages that have already been executed by the island before the snapshot was created are discarded by the controller and other messages are added to the message queue if it does not contain them yet. Afterwards, normal operation can be resumed by starting the new island replica's message processing.

As individual participants are not required for the continued execution of other island replica's, participants are free to leave the island at any point and, for the same reason, the TeaTime architecture is resistant to failures of individual participants. Participants that were intermittently disconnected from the router, however, have no other way of rejoining the island but to request a new snapshot of the island state again, because messages may have been lost in the meantime. Furthermore, the message router itself presents a single point of failure in TeaTime's architecture, as we will discuss in Section 4.5.2.

4.2.2 Configuring the Distribution Boundary

As described in Section 3.3.3, it should be possible to configure and encapsulate the distribution boundary in the interaction with shared objects within their implementation or deployment, so that users of reusable shared objects can then interact with shared objects as if they were local and leave the responsibility of deciding when to cross the boundary to the shared object's implementation.

Therefore, Archipelago/S extends the Island/TeaTime technology to allow encapsulating this boundary within the implementation of a shared object: The developer of a shared object is able to specify parts of its functionality that are to be executed before crossing the boundary, that is, whose effects are not replicated. We call this part of the functionality the shared object's *local side*, and the other parts its *shared* or *replicated side*. The developer can use the local side to handle the issues related to crossing the boundary or to implement functionality that can be executed without crossing the boundary, that is, without replicating its effects. The local side of a shared object can also be used to maintain references to local objects outside the shared space.

We implement the division of local and shared functionality of a shared object with two new concepts, called boundary objects and auto message makers, which we describe in Section 4.4.

4.3 Implementation of Croquet's Replication Technologies

In this section, we describe select details of the implementation of the replication technology of the Croquet system. We chose to include these details in this thesis, because Croquet's implementation is in many places only sparsely documented and we had to recover the intent and inner workings behind the implementation through careful source code analysis. We think our insights may help future continued development. Furthermore, knowledge of some details of Croquet's implementation is important to discuss our changes to Croquet's technologies for use in Archipelago/S.

The implementation of Croquet's replication technology is split into two parts, known as the *Islands* and *TeaTime* technologies. The Islands technology by itself supports the consistent separation of local segments of

Squeak images, called islands, and a way to intercept messages sent to objects inside another island by means of automatic object proxying. These islands are separated parts within the local runtime system of a single participant. The TeaTime technology builds upon the Islands technology and adds mechanisms to replicate local islands between multiple participants. For this purpose, it uses the message interception instruments of the Islands technology to replicate messages sent to objects within an island via the message router to all participants of a replicated island.

4.3.1 Islands: Separate Image Segments

The separation of parts of the Squeak image into local islands has two main purposes: First, it allows messages sent to objects within an island to be intercepted, and second, it allows an island to be snapshot at any point in time. For TeaTime, intercepting messages is used to replicate messages and snapshotting islands is used to create new island replicas and allow new participants to join a replicated island.

The Islands technology associates every object with a local island that it resides in. This means, even objects local to a single participant reside within a default island in the local Squeak runtime system. Objects shared with other participants reside in another island in the runtime system, which is replicated by means of the TeaTime technology described in the next subsection.

An island is represented by an instance of the `Island` class and isolates different sets of interacting objects from each other. Each object resides in exactly one fixed island, which is the island that it was created in. This means, it is not possible for the object to migrate to another island without creating a copy of itself. In order to keep up isolation between islands, objects in different islands cannot directly refer to each other. Instead, they use proxy objects called *far references* or *FarRefs*, represented by instances of the `FarRef` class, to refer and send messages to objects in other islands. Objects can obtain FarRefs to objects within another island through return values of message sends to objects in the other island, or by requesting a reference to a named object from the other island. An example for the interaction with objects in other islands is shown in Listing 4.1.

When an object sends a message to a FarRef, the arguments of this message cannot be passed directly to the receiver object represented by the FarRef, since it resides in a different island. They either have to be wrapped

```
| island list |
"Create new Island instance."
island := Island named: 'example'.

"Create a new object of type OrderedCollection inside the island."
list := island new: OrderedCollection.
"Note: list is now a FarRef pointing to the new object."

"Add a value to the list."
list add: 42.

"Obtain and print the value from the list."
Transcript showln: 'The value in the list is: ', list first.
```

Listing 4.1: Interaction with an object residing in a different local island. Even though the external object is proxied by a FarRef, the interaction with it is not different from the interaction with a common local object.

into FarRefs pointing back from the receiver's island to the argument objects in the sender's island, or have to be copied into the receiver's island and those copies have to be passed instead. The same applies for the return value passed back from the receiver to the sender object. The decision whether an argument or return value object is copied or passed as FarRef is made depending on the type of the passed object.

Generally, argument and return value objects of such messages can be passed in four different ways: by reference, by copy, by clone, and by identity. Passing by reference obtains a FarRef to the object from its island and passes the FarRef instead; this is the default behaviour.

Passing by copy creates a copy of the object within the receiver's island and passes a direct reference to this copy. Copying objects instead of passing references can be advantageous in terms of performance: It can be faster to copy a small, short-lived object and allow a direct interaction with this copy instead of a proxied interaction with the original object. However, copying the object separates the state of the original object from its copy, which means that later changes to the copy are not reflected in the original object and vice-versa. Typical objects passed by copy are Strings and Symbols. When an object is passed by copy, nested references within the copied object are passed according to their own type.

Passing by clone is like passing by copy, but allows a single object that appears at multiple locations inside an object reference graph to be copied multiple times, that is, it is a variant of passing by copy that is not identity-preserving. Passing by clone can be faster for a small object than passing by copy, because references to the object do not have to be

tracked during the copying process. It does, however, result in a different object reference graph, may be less efficient for complex object graphs with multiple references to the same object, and can also lead to infinite loops if the object refers back to itself (directly or indirectly). In Croquet, passing by clone is used, for example, for Point or Rectangle objects.

Some objects can be passed by identity, that is, a direct reference to them can be passed to objects inside another island, and no copying or proxying is necessary. There are two kinds of objects that use this policy: First, basic, immutable objects, such as Integers or Booleans. It makes sense to pass these objects directly, since they do neither contain references to other objects nor can they change their value. Second, classes and process-related objects cannot be passed as proxies or copies, as the Squeak VM requires direct interaction with these.

These passing policies do not yet have anything to do with sharing objects or islands between multiple participants. They only serve for the automatic and transparent isolation of different local islands, that is, segments of the Squeak image. This isolation is imperative for creating a snapshot of all objects within an island, which can allow copying an island to different participants. In order to efficiently create such a snapshot, objects inside the island need to have as few references to objects outside the island as possible. Objects that are contained within the island can simply be serialized and written to the snapshot. However, outside objects referenced by objects inside the island have to be handled differently. It is important that these references remain valid when a new island copy is created by loading the snapshot. Because of the parameter passing policies, the only kinds of outside objects directly referenced from objects within an island are FarRef instances and objects that are passed by identity, as well as compiler literals that may be shared between different islands. Therefore, when creating and loading the island snapshot, FarRefs, pass-by-identity objects, and compiler literals are handled separately: They are serialized in specialized ways that allow reloading them. For example, a class is serialized as class name and version number and de-serialized by looking up the class object corresponding to the serialized name and version number. By default, FarRef objects are not serialized at all but replaced with nil values. TeaTime extends this mechanism and allows FarRef serialization so that FarRefs pointing to shared objects can be serialized and snapshot, as described in the next subsection.

More details about the implementation of Islands and FarRefs are documented in the source code and class comments of the Islands technology.

4.3.2 TeaTime: Computational Replication for Islands

TeaTime extends the Islands technology with functionality for replicating an island and synchronizing the objects contained within it between multiple participants. This is primarily accomplished by extending the `Island` and `FarRef` classes with the message replication functionality described in the previous section. These extensions are contained in the subclasses `TIsland` and `TFarRef`. The island replicas residing at a participant, that is, TIslands instances, are managed by an instance of the `CroquetHarness` class. A CroquetHarness can create new replicated islands and manages router and controller setup and connection and island discovery in the network. We first describe the implementation of the message replication with TIslands and TFarRefs and then describe the inner workings of the CroquetHarness.

Message Replication with TIslands and TFarRefs

TIslands. The `TIsland` class is a subclass of the `Island` class. It adds support for an island controller, scheduling replicated messages within a message queue as described in the previous section and for the processing of this queue, as well as keeping the current island time. The TIsland class also replaces the way that an island creates a FarRef to an object inside the island: Instead of creating a `FarRef` instance, it creates a `TFarRef` instance.

TFarRefs. TFarRefs are derived from the `FarRef` class and intercept and replicate messages to shared objects. Each object in a TIsland can be exported, that is, made accessible from outside the island, by means of a TFarRef; however, only a single TFarRef for each object can exist. Therefore, the TIsland keeps a map of all exported objects and the TFarRefs used to export them. When a TFarRef to an object inside the replicated island is required, for example, to return a reference to an object in the island as a return value of a replicated external message, a lookup in this map is performed. If an existing TFarRef for the object is found, this TFarRef is returned; otherwise, a new TFarRef is created and added to the map.

Because an exported object can only have a single associated TFarRef, TeaTime can associate an identifier with each TFarRef that also functions as the identifier of a shared object. The TIsland also keeps a map associating these identifiers with the exported objects. When a TFarRef is exported during the snapshotting of an island or if it is passed as an argument to a replicated message, it can thus be serialized by writing this identifier and de-serialized by looking up the object with the corresponding identifier. This identifier can even be used to refer to shared objects that have not been created yet, such as the future return values of replicated messages, as we will describe later.

A TFarRef actually allows the user of a shared object to either send a message to the shared object in a way that will replicate it or in a way that executes it only on the local object copy without replication. In Croquet, the latter is only used for special cases, such as the local rendering of shared objects. We illustrate the interaction with a shared object inside a replicated island in Listing 4.2. It shows both how to send replicated and locally executed messages to the shared object.

Interaction with TFarRefs. Replicated messages can be sent to an object represented by a TFarRef by sending the message `#future` to the TFarRef followed by the message to be replicated. This is shown in lines 7, 13 or 16 in Listing 4.2. As described in the previous section, such a replicated message is then wrapped into a message object and sent via the local island controller to the router, which then forwards the message to all island replicas to be scheduled for execution. In TeaTime, this happens asynchronously: When a replicated message is sent to a TFarRef, the TFarRef sends the message to the controller, who then queues the message for sending to the router and returns program flow via the TFarRef back to the sender of the replicated message. Because of this, the effects and return value of a replicated message are not visible to the sender immediately after the message is sent.

To enable the sender to wait for the effects, obtain the actual return value, and send further messages to the return value, it is supplied with a new TFarRef acting as a promise for the return value of the replicated message. This TFarRef is assigned a new identifier, which is also attached to the replicated message before it is sent to the controller. The result of the message will later be stored under this identifier inside the TIsland when the message is processed by the island replicas. To send further replicated

```
1  | harness island list number |
2  "Create new CroquetHarness and replicated TIsland instance."
3  harness := CroquetHarness new.
4  island := harness createIslandNamed: 'example'.
5
6  "Create a new object of type OrderedCollection inside the island."
7  list := island future new: OrderedCollection.
8  "Note: list is now a TFarRef referring to the replicated object."
9  "Note also: The replicated object may not exist yet,
10  since the future call is asynchronous."
11
12 "Add a value to the list."
13 list future add: 42.
14
15 "Obtain the value from the list via a replicated message."
16 number := list future first.      "value is now a TFarRef."
17 number wait.                      "Wait for resolving of the TFarRef."
18 number := number value.           "Retrieve the actual number value."
19
20 Transcript showln: 'The first value in the list is: ', number.
21
22 "Alternative: Add a value replicated and read it locally."
23 (list future add: 23) wait.       "wait until the value was added."
24 number := list send:
25             [:obj | obj second].  "send the message #second locally."
26
27 Transcript showln: 'The second value in the list is: ', number.
```

Listing 4.2: Interaction with an object residing in a replicated island. The external object is proxied by a TFarRef, and the user of the object has to send explicitly replicated or local messages to it and handle the return values accordingly.

messages to the return value represented by this TFarRef, it is not necessary to wait for the completion of the execution of the initial replicated message: Since any later replicated message will be executed after the initial message, the return value will be available before a later replicated message is executed. This can be observed between lines 7 and 13: The variable `list` holds a TFarRef to a shared object, which is asynchronously created in line 7, but we can send another replicated message to it in line 13, without waiting for it to be resolved.

In order to obtain the actual return value or send locally executed messages to the return value of a replicated message, however, the sender needs to wait for the execution of the replicated message to be completed. This can be accomplished by sending the message `#wait` to the TFarRef representing the return value, as shown in lines 17 or 23. The wait message only returns once the replicated message was executed by the local TIsland replica and, therefore, ensures that the effects are visible to the sender and that the TFarRef now points to a result value object that actually exists.

The sender can then send the `#value` message to obtain the result value, as shown in line 18.

Local messages can be sent to a shared object represented by a TFarRef by sending the message `#send:` to the TFarRef with a block as its argument, as shown in lines 24 and 25. The block is evaluated with a different proxy to the shared object as its first argument. This proxy relays any message sent to it directly to the shared object. The block can thus perform any operation on the shared object, and return the result of such operations back to the sender of the `#send:` message. Because local messages are executed synchronously, it is not necessary to wait for the result of a local message, as would be the case for a replicated message, which can be observed in line 27: The result of the `#send:` message can be used directly for printing on the Transcript. However, local messages can only see those effects of replicated messages that have already been processed by the local TIsland replica. For example, the local message in lines 24 and 25 needs to wait for the side effects from the replicated message in line 23 to take place, which is why the message `#wait` is sent to the result of that message in line 23. Local messages are not executed concurrency synchronized with the processing of replicated messages, which means that local messages may be executed on an inconsistent state of a replicated object. The implementation of local message sends to a TFarRef is essentially the same as sending messages to FarRefs and uses the same argument and return value passing policies for messages sent to objects in different islands.

What can be taken away from the use of TFarRefs as illustrated in Listing 4.2 is that the interaction with shared objects in Croquet is fundamentally different to the interaction with local objects. Users of a shared object have to decide when to send a replicated message and when to send a message that should be executed locally. Furthermore, they have to take care when handling the return values of such messages: For example, when the return value of a replicated message is of a basic type, the users may need to wait for the execution of replicated messages and to explicitly obtain the actual return value from the TFarRef by sending the `#wait` and `#value` messages. However, when sending further replicated messages to the return value, neither of these are necessary.

Parameter Passing. The parameter passing behaviour for replicated messages is different from the policies that the Islands technology uses for locally executed messages. Arguments to replicated messages are always

passed as deep copies, except for arguments that are TFarRefs pointing to shared objects. Such a TFarRef argument is resolved to an actual reference to the shared object it represents before the message is scheduled to be processed by the TIsland. This means, references to shared objects that are passed as arguments continue to refer to the same shared object during the execution of the message. Any other argument is passed by serializing the argument object, sending this serialized copy in the message object and de-serializing it before the message is scheduled to be executed in each Island replica. This essentially means that identical deep copies of the argument object are created and used as argument to the execution of the replicated message in each participant.

Active Objects. As described in Section 4.2, shared objects can be active objects and send internal replicated messages to themselves or each other to exhibit behaviour. TeaTime implements this by allowing shared objects to send the message `#future:` followed by the actual message to be sent to themselves or any other shared object. This message takes as argument an integer number that denotes the time delay in milliseconds from the current island time at which the message should be scheduled for execution. The message is wrapped into a message object and directly added to the message queue of the shared object's island. In the following example, this mechanism is used to schedule sending the `#step` message to the receiver object at 100 milliseconds after the current island time:

```
(self future: 100) step
```

Island Setup with the CroquetHarness

Croquet uses the `CroquetHarness` class to setup and connect the infrastructure behind replicated island instances. Each participant uses a local CroquetHarness instance. In Croquet, each harness only supports a single currently active TIsland instance, but participants can use multiple harness instances to join multiple shared islands at the same time.

The CroquetHarness allows to create new replicated TIsland instances, announce these on the local network and find and join replicated islands created and offered by other participants. Therefore, it combines the tasks of setting up and managing local island controller instances, local message

routers, local island replicas, and network contact points for broadcasting and receiving island announcements.

Applications in Croquet are 3D-rendered worlds that are fully contained inside a replicated island and in which users can interact with each other and with other objects in the world. Users can also use links to switch from one world to another. These links are called postcards to special entry points to Croquet Worlds, which are known as portals. To support these concepts, the CroquetHarness also integrates a rendering loop and forwarding of device events into the replicated world, as well as the resolving mechanism for postcard links. For performance reasons, this functionality is tightly intertwined with the message processing and joining of islands: Rendering, for example, occurs in between message processing steps of the local island controllers, in order to make sure that a consistent state of the 3D world is rendered. Listing 4.3 shows an example for a typical setup of a world in a new replicated island with the help of a CroquetHarness. By default, the participant creating a new island with the harness functions as the message router for the island. A similar mechanism can be used to join existing replicated islands.

4.4 Changes to Islands and TeaTime for Archipelago/S

As mentioned in Section 4.1.2, we separated the Islands and TeaTime technologies from the Croquet system for use in Archipelago/S. Croquet applications are replicated collaborative 3D worlds inside an island, and Croquet's implementation of Islands and TeaTime is specialized towards this use case, as described in the previous section. We untangled Croquet's replication technologies from functionality particular to this use case, ported them to Squeak 4.5 and extended them with additional advanced and previously missing functionality.

In Croquet applications, only small terminal-like functionality is not replicated within the island, such as rendering the worlds, handling and passing on device events, or setting up applications. In Archipelago/S, the architecture of applications is more general, and applications are comprised of computationally different parts distributed amongst multiple participants. For Archipelago's use case, therefore, the interaction between local and shared objects is more important than for applications in Croquet:

```
| harness pc sync entry island |

"Create a new harness instance."
harness := CroquetHarness new.

"Add a graphical overlay that is rendered on top of the world."
harness addOverlay: self.

"Create a new island and world inside it."
island := harness createIsland: ExampleWorld named: 'Example World'.

"Register a new portal to the world and obtain a postcard to it."
entry := harness registerPortal: #mainEntry in: island.
pc := entry future postcard.   "pc now is a TFarRef to the postcard."

"Add a hook to execute when the postcard is created."
pc whenResolved:[
    "Tell the 3d view managed by the harness to follow the postcard link."
    sync := harness viewPortal future postcardLink: pc value.

    "When the link is resolved, tell the harness to start the rendering
    and message processing for the island."
    sync whenResolved:[harness startRender].
].

"Wait for the postcard to be created."
pc wait.

"Now, a new 3D world of the type ExampleWorld was created and the user
has joined the world through the portal #mainEntry. Rendering has
commenced and the user can interact with objects inside the world."
```

Listing 4.3: Setting up a Croquet world with the CroquetHarness.

Application developers in Croquet usually only develop objects that will be shared and their interaction with other shared objects; application developers in Archipelago develop both shared and local objects that interact with each other.

As described in the previous section, the interaction between local and shared objects in Croquet is very different from the interaction between local ones, and the distribution boundary is fully exposed to the user of a shared object. We argue that this way of interaction is inconvenient for users of shared objects and does not allow the implementation of convenient, reusable shared object structures. Therefore, we introduce mechanisms that allow more convenient interaction with shared objects in islands and enable the encapsulation of the distribution boundary within the shared objects.

In this section, we first describe the separation of the replication technologies from the Croquet system and their extensions with additional

functionality. Subsequently, we describe the implementation of the mechanisms that allow the encapsulation of the distribution boundary within shared objects for more convenient interaction with shared objects.

4.4.1 Islands and TeaTime in Archipelago/S

For use in Archipelago/S, we untangled the Islands and TeaTime technologies from functionality specific to Croquet's 3D world applications; this includes removing all dependencies to worlds, portals, rendering, and device events. Most of this functionality was contained in the CroquetHarness class and interleaved with TeaTime functionality, as described in Section 4.3.

Croquet's Islands and TeaTime implementations were also missing certain features and contained a few bugs, which we added, respectively fixed, in the process of separating and applying the technologies for use in Archipelago/S.

We first describe our modifications of the CroquetHarness, then explain the details of some other extensions and bugfixes we implemented.

Changes to the Harness

For Archipelago/S, we replaced the `CroquetHarness` class with the new class `ArchipelagoHarness`. We copied and extended a lot of the original functionality of the CroquetHarness, but also removed equally much: We kept all TeaTime-related functionality, but removed all functionality related to Croquet's 3D worlds. The latter includes the removal of worlds, portals, 3D overlays, device events, OpenGL, and other rendering dependencies. For this purpose, we also had to untangle the message processing of replicated islands managed by the harness from the rendering process. Furthermore, we re-purposed postcard links as identifiers of shared islands: They now contain information specifying which participant to contact to join a particular island and are used during the announcement of available islands.

We further extended the ArchipelagoHarness with new functionality. The harness now supports multiple active islands at the same time, which reduces the overhead of using multiple islands within a single application. Furthermore, we extended the discovery of available islands to support timeouts, the simultaneous search for multiple islands, and events for

newly discovered islands. The new harness implementation also allows to leave previously joined islands and can handle unexpected network interruptions, such as the break-off of the connection to the island's message router. It also adds a listener mechanism for objects outside the space to detect and handle island shutdowns caused either by network interruptions or by leaving the island. The ArchipelagoHarness can now be used to create, join, or leave shared islands as shown in Listing 4.4. This additional functionality is necessary since Archipelago's application setup is intrinsically more dynamic: Parts of a distributed application can join or leave the system during its execution.

```
| harness island island2 island3 names list |

"Create a new harness instance."
harness := ArchipelagoHarness new.

"Create a new island."
island := harness createIslandNamed: 'example'.

"Alternative: Search for an existing island created by another participant."
island2 := harness findIslandNamed: 'foobar'.

"Alternative: Obtain list of discovered islands and join first."
names := harness discoveredIslandNames.
island3 := harness findIslandNamed: names first.

"Now, the application can create or move objects into the islands, such as:"
list := island future new: OrderedCollection.

"Leave islands again."
harness
    leaveIsland: island;
    leaveIsland: island2;
    leaveIsland: island3.
```

Listing 4.4: Using the ArchipelagoHarness to create, join, and leave islands.

Other Extensions and Bugfixes

Our other modifications to Islands and TeaTime include storing blocks in shared objects, fixing issues of multi-threading, allowing the nesting of identifiers in shared objects, additional parameter passing behaviours, and blocking operations on shared objects.

Storing Blocks in Shared Objects. Croquet's Islands/TeaTime implementation does not allow shared objects to store code blocks in their in-

stance state. In Squeak/Smalltalk, however, code blocks are often used, for example, in observer situations as event handlers. We added functionality that allows storing such blocks in the instance state of shared objects. In particular, this involves support for serialization and de-serialization of code blocks and compiled methods in the `IslandWriter` and `IslandReader` classes that are used to create snapshots of islands. Because the blocks are serialized, our implementation is limited to blocks that only reference static variable values, as discussed in Section 4.5.2.

Multi-Threading Issues. The original Islands implementation does not support manual forking of threads inside a shared island. This is necessary, for example, to notify asynchronous listeners residing outside the island from within the island. Therefore, we modified the forking behaviour so that a forked thread remains local to the island that it was created in. We also fixed a concurrency bug in the `#wait` message implementation of the `TFarRef` class. This bug was caused by missing concurrency synchronization, that is, mutual exclusion, during the resolving of the represented object and could manifest itself as a deadlock situation.

TFarRef Identifier Nesting. As described in Section 4.3, return values of replicated messages sent to TFarRefs are passed according to the parameter passing behaviour of the Islands technology. This means, objects of some types, for example, arrays, are passed as copy or clone and can contain references to shared objects that are passed by reference, that is, by wrapping into a new TFarRef. In the original implementation of TeaTime, these nested TFarRefs in objects passed by copy or clone did not have an associated identifier. Because of this, further replicated interaction with the objects represented by these nested TFarRefs was not possible. We saw two options to circumvent this problem: First, we could register each nested object that a TFarRef is passed out of the island for as an exported object within the TIsland. While this may be a clean way, it can create a large number of unnecessarily exported objects cluttering the island's map of exported objects. Because this map lives as long as the island exists and is not garbage collected, its unnecessary enlargement is a big disadvantage. The second option, which we also chose to implement, is to allow the identification of nested objects by their position inside the parent's object. This, of course, can become a problem if the parent's object is manipulated

after creation of this nested identification. We currently do not allow such manipulation, and raise an exception if a manipulation is detected. We expect this approach to be appropriate for most use cases, as objects that are passed by cloning or copying and contain references to other shared objects are usually not changed after passing them out of the island. However, we are still investigating whether the previously presented, cleaner, but less efficient approach of exporting every referenced object is better suited for a user-friendly platform implementation.

Parameter Passing. During our development, we discovered the need for an additional parameter passing behaviour next to passing by reference, copy, clone, or identity: passing by deep copy. This new behaviour modifies passing by copy so that all objects that are referenced by the passed object and are typically passed by reference are also passed by (deep) copy. This allows application developers to wrap shared objects into container objects that are always passed as a single coherently copied entity.

Furthermore, our implementation of Islands also allows for passing of message maker objects. Message maker objects are proxy objects to shared objects created, for example, by sending the `#future` message to a TFarRef. We allow to pass such message maker objects in the same way as TFarRefs to shared objects: When passed into the shared island, they are replaced with a common object reference to the shared object. This behaviour is particularly important for the passing of auto message makers, which are described in the next subsection.

Blocking Operations. The original TeaTime implementation from Croquet does not support the replicated execution of messages on shared objects that could block, that is, wait for an external event to be signaled by another replicated message execution. The problem here is that a replicated message that blocks, for example, by waiting on a semaphore, will also block the message processing thread and prevent further processing of later messages. This will lead to a deadlock, because the replicated messages that could wake the blocking message up, for example, by signalling the semaphore, will never be executed. However, we discovered the need for blocking replicated messages during the development of our tuple space—an example for a replicated blocking message is its take operation, which we describe in Section 5.1.

Therefore, we extended the TeaTime implementation with support for such blocking replicated messages. Our approach is based on the following idea: When the message initiates the blocking, we suspend its execution. We then continue the processing of other messages in the message queue until the blocking message is woken up by one of these executed messages. When this happens, we resume the execution of the blocked message at the point where it left off and complete its execution, before continuing the execution of the message that caused the wake up.

We implemented this mechanism by means of a custom semaphore implementation in the `TSemaphore` class, which has to be used by shared objects for implementing the blocking behaviour. When a replicated message sends the `#wait` to a TSemaphore, the island saves the state of the message execution as replicated state in the island. We make use of Squeak's MethodContexts to extract the state of the message execution by saving the methods that were active when the semaphore wait occurred, as well as their corresponding receiver and stack values (as identifiers of replicated objects), stack pointers and program counters. The state is associated with the identifier of the TSemaphore that was waited on and saved into a dictionary in the island's replicated state.

When another replicated message later sends the `#signal` message to the TSemaphore, the island restores the state associated with this TSemaphore. This involves recreating the MethodContexts that were active when the wait occurred and resuming their execution until the previously blocked message has finished execution. We also support further subsequent waits in the execution of the same message, which are implemented in the same way.

Our current implementation of this mechanism is limited in that we only allow a single concurrent execution to wait on the semaphore at a time and only allow other replicated messages to wake up a blocked replicated message. We believe that the first limitation can be overcome by introducing a queuing mechanism for TSemaphore waits; this, however, requires significantly more work because of potential dependencies between waiting executions. The latter limitation, however, cannot easily be overcome, because it ensures that the resumption of the blocked message execution will be caused at the same point in the message processing at all participants.

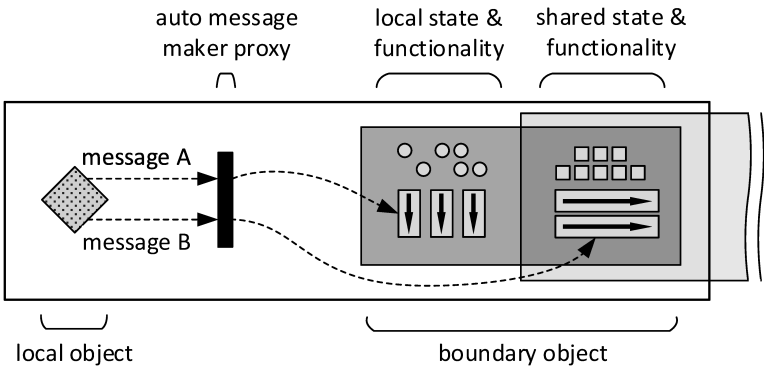

Figure 4.3: Boundary objects and auto message makers. Message A is executed on the local side of the shared object, message B is replicated.

4.4.2 Encapsulating the Distribution Boundary

Interacting with shared objects through TFarRefs requires the users of the shared object to specify the distribution boundary: The users determine which messages are sent locally and which are replicated. They further have to determine whether the return value of a replicated message requires sending the `#wait` and `#value` messages to access it, for example, because it is of basic type. In order to conveniently use shared objects, Archipelago/S adds mechanisms for encapsulating the distribution boundary within the implementation of the shared object itself. Thereby, we make it possible to hide the boundary from users of shared objects, which allows to implement conveniently accessible and reusable object structures in the shared space. We propose two new concepts that implement these mechanisms: boundary objects and auto message makers, illustrated in Figure 4.3.

Boundary Objects

Boundary objects are shared objects that encapsulate the distribution boundary. We achieve this by splitting the state and functionality of a boundary object into two parts, which we call the local and shared side of the object. The local side can contain functionality that is executed without or before crossing the distribution boundary and the shared side contains functionality that is executed after crossing the boundary. The local side

can also be used to store references to objects outside the replicated island and for interaction with them.

In our current implementation, the two sides are not encapsulated from each other, that is, local functionality can also access shared state and functionality, and shared functionality can also access local state and functionality. This allows, for example, to read shared immutable state by sending a local message or to use the local side of a shared object to store references to observer objects outside the shared island and send messages to these objects as side effects of replicated messages. However, as local functionality is not replicated, it must not modify any shared state—if it does, it risks inconsistencies between replicated copies of the shared object. Currently, we leave the responsibility for ensuring this to the developer; for a discussion of this issue see Section 4.5.1.

```
1  ObservableList>>items
2    ↑ items ifNil: [items := OrderedCollection new]
3
4  ObservableList>>listeners
5    <storeLocally>
6    ↑ listeners ifNil: [listeners := OrderedCollection new]
7
8  ObservableList>>add: anObject
9    self items add: anObject
10   self listeners do: [:each | each value: anObject]
11
12 ObservableList>>addSum: aCollection
13   <executeLocally>
14   |sum|
15   sum := aCollection reduce: [:a :b | a + b].
16   self replicated add: sum
17
18 ObservableList>>addListener: aListener
19   <executeLocally>
20   self listeners add: aListener
21
22 ObservableList>>values
23   <executeLocally>
24   ↑ Array withAll: self items
25
26 ObservableList>>size
27   <executeLocally>
28   ↑ self items size
```

Listing 4.5: Example for labelling of local state and functionality in a boundary object.

To implement a shared object as a boundary object, the developer explicitly labels local state and functionality by annotating methods and field variable accessors with pragmas. State and functionality that is not labeled

as such is assumed to be shared. An example for this labelling is shown in Listing 4.5: Instances of the ObservableList class are shared boundary objects, which store a replicated OrderedCollection containing items and a participant-local OrderedCollection of listeners that are notified when new items are added to the ObservableList. Users can also obtain the current size of the list or a copy of the currently stored items in the list.

The listeners are stored locally, denoted by the pragma `<storeLocally>` in line 5, because they are intended to be objects outside the replicated island and references to them are only valid in a single participant's runtime system. The method `#addListener:` adds new external listener objects to the `listeners` collection and, because of this reason, is executed locally. The local execution is denoted with the pragma `<executeLocally>` in line 19. However, the listeners are also accessed and interacted with by the implementation of the replicated message `#add:`. This is an example for shared functionality accessing local state.

The listing also shows the implementation of the locally executed message `#addSum:`, which takes a Collection of numbers and adds the sum of the numbers to the list. We first calculate the sum locally, then send the `#add:` message to the receiver object as a replicated message. Thereby, we explicitly indicate that the distribution boundary should be crossed: We use the message `#replicated` to denote that the following message `#add:` is to be sent replicated, as shown in line 16.

The messages `#values` and `#size` are implemented as local ones, even though they access shared state. This means, they may not return values corresponding to the most current state, since not all previous replicated messages may have been executed in the local island replica yet. However, our implementation of boundary objects and auto message makers ensures that local messages always operate on a consistent version of the replicated state. The advantage of executing these messages locally is, first and foremost, performance: We can save the overhead of replicating the messages via the router to all island replicas. In a real application, whether such an access can be executed locally will depend on the nature of the replicated data and its use. In Section 5.1, we describe our tuple space implementation in Archipelago/S as another real-life example for such accesses.

Boundary objects are implemented as subclass of the `BoundaryObject` class. This class provides additional behaviours that are executed during method compilation of its subclasses, which collect all methods and in-

stance variables that belong to the local side of the object and store their names in lists with the compiled subclass. These lists are later used for two purposes: Locally stored state of a shared object has to be exempt from snapshotting and replication of the object. Therefore, the instance variables denoted as `storeLocally` in the lists are cleared when a snapshot of the island is created and transferred. Furthermore, the lists are also used for the efficient lookup of the strategy to be used when a certain message is to be sent to a shared object, as we will explain in the following.

Auto Message Makers

Auto message makers form the counterpart to boundary objects and function as additional proxy objects around TFarRefs. They allow users of the shared object to interact with it as if it was an ordinary local object: Users can send messages to the auto message maker that are then relayed to the shared object either as replicated or local messages, depending on the object type and the labelling of the method invoked by the message. Auto message makers can also be used with simple shared objects that are not special boundary objects: In this case, all messages sent to the object are replicated.

Furthermore, auto message makers send all messages synchronously and perform a type-based return value wrapping of replicated messages: The return value of a replicated message is automatically either wrapped into an auto message maker itself if it is a shared object that should be passed by reference, or unwrapped and passed back by value if it is of a type that is passed by copy, clone, or identity.

In consequence, users of shared objects that are accessed through an auto message maker no longer have to use the special interface of TFarRefs with `future`, `wait` and `value` messages, and the interaction with the shared object is just like the interaction with a local one, as illustrated in Listing 4.6. The listing shows the creation of a new ObservableList boundary object in a replicated island and the interaction with it through an auto message maker. Therefore, the TFarRef representing the shared object is wrapped by an auto message maker by sending the message `#autoFuture` to the TFarRef in line 11. The return value of this message send is then used for later interaction with the object. Because the auto message maker also executes replicated messages synchronously, later locally executed messages, such as the `#size` and `#value` messages in lines 22 and 23, are

ensured to see the side effects of previously sent replicated messages from the same process thread, such as the `#add:` and `#addSum:` messages in lines 18 and 19.

```
1  | harness island list size values |
2  "Create new harness and replicated island."
3  harness := ArchipelagoHarness new.
4  island := harness createIslandNamed: 'example'.
5
6  "Create a new shared ObservableList."
7  list := island future new: ObservableList.
8  "Note: list is now a TFarRef referring to the replicated object."
9
10 "Wrap list with an auto message maker."
11 list := list autoFuture.
12
13 "Add a listener to the list - message is sent locally."
14 list addListener: [:newValue |
15      Transcript showln: 'New value added: ', newValue asString].
16
17 "Add values to the list - messages are sent replicated."
18 list add: 42.                  "Output: 'New value added: 42'"
19 list addSum: {1. 2. 3. 4. 5}.   "Output: 'New value added: 15'"
20
21 "Obtain the size and values from the list - messages are sent locally."
22 size := list size.
23 values := list values.
24
25 Transcript showln: ('There are ', size asString,
26                     ' values in the list: ', values asString).
```

Listing 4.6: Interaction with a shared object through an auto message maker. The shared object is accessed as if it was local: All message sends are synchronous and automatically either replicated or executed locally.

Auto message makers implement a proxied message send as follows: When a message is sent to an auto message maker proxy, it obtains the type of the shared object represented by the TFarRef the message maker wraps. If this type is a subclass of the `BoundaryObject` class, it then uses the lists stored with the class to lookup whether the message is to be sent locally or replicated. If it is a local message, it sends the message via a `#send:` message to the TFarRef, if it is a replicated message, it sends it via a `#future` message to the TFarRef. In the replicated case, the auto message maker then waits until the result value is resolved; in the local case, this is not necessary. In either case, it returns the result as newly wrapped shared object if it is a pass by reference object, or as unwrapped object if it is a pass by copy, clone, or identity object.

The auto message maker also ensures that locally executed messages are concurrency synchronized with each other and with the execution of replicated messages inside the island. There is one caveat here, though: If a local message blocks, the execution of replicated messages is also blocked because of the concurrency synchronization. To solve this issue, we also added a special mechanism to support blocking local messages. Therefore, we require blocking local operations to use the `TSemaphore` class described before to implement the blocking, but we handle the wait on a TSemaphore differently when it is sent from within the execution of a local message. In this case, we simply exit the concurrency synchronization, allow other local or replicated messages to be executed and wait for a signal on the same TSemaphore by one of these messages. When this happens, the local message reenters the synchronization and resumes its execution.

4.5 Discussion

Archipelago/S provides a prototype implementation of the Archipelago platform design based on replication of shared objects between participants. Replication of shared objects allows Archipelago/S to tolerate failures of individual participants and enables peer-to-peer application architectures.

The use of the Islands and TeaTime technologies makes sharing objects simple and efficient: Objects are deployed into an island, replicated at all participants and held consistent by replicating computations on them. Generally, the technologies allow any object to be shared into an island.

Furthermore, our auto message maker implementation allows users of shared objects to interact with them as if they were local objects, hiding the distribution boundary from them. Developers of advanced shared objects can use the boundary objects mechanism to define a shared object's local and shared sides and to encapsulate and configure the distribution boundary, which enables the implementation of conveniently reusable coordination mechanisms and data structures in the shared space, as we will show exemplarily in Chapter 5.

As described in Section 4.2, Archipelago/S fulfills the responsibilities that the Archipelago platform design delegates to its implementation: consistency of shared objects, independence of shared objects from the avail-

ability of individual participants, registration and naming mechanisms, and concurrency synchronization of operations on shared objects.

However, there are certain implementation choices that we are not fully convinced of. We are still evaluating these choices and continue to look for alternatives, as we discuss below. Besides this, our implementation also has a few disadvantages and limitations, which we discuss subsequently.

4.5.1 Open Questions

During the implementation of Archipelago/S, we made a few choices that we feel need further evaluation and investigation.

Distribution Boundary

We have identified the need for the encapsulation of distribution boundaries for shared objects. Therefore, our implementation of boundary objects makes this boundary explicit within the implementation of a single shared object. This gives the developer of a shared object a powerful tool to implement the local side of a shared object, as it can freely access the replicated state of the object; but at the same time, this is also dangerous: functionality on the local side of the object is in no way prevented from modifying shared state, which would undermine the consistency between the object replicas. We currently investigate methods that could prevent such modifications by means of the platform infrastructure, such as statically or dynamically analyzing local functionality for such breaches or sandboxing local method executions, for example, by executing them in a transactional context.

An alternative to these approaches would be to divide a shared object into two objects, a local proxy object and the shared object. Then, the local object would only be allowed to access shared state by means of replicated interaction with the shared object, which comes with the disadvantage that any non-modifying interactions on shared state have to be replicated and replicated state may have to be additionally cached in the proxy object for performance reasons. This approach is similar to the Half Object Plus Protocol pattern, which divides a shared object into two halves residing in different address spaces and a protocol in between them [12].

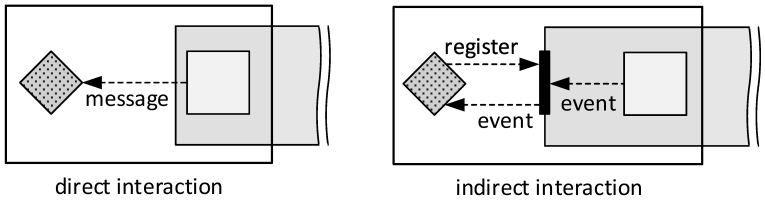

Figure 4.4: Direct vs. indirect interaction with local objects. We show the messages exchanged.

Interaction with Local Objects

A similar issue exists for the interaction of shared objects with local objects. For Archipelago/S, we chose to allow a direct interaction with local objects from within shared objects, as used for the interaction with listeners in the example in Listing 4.5. This direct interaction gives shared objects a lot of flexibility in the interaction with local ones, but also incurs a breach between the shared and local sides of a shared object: The shared side needs to access locally stored references and send messages to those objects. This makes it possible that replicated functionality can send messages to local objects and use their return values—the developer has to make sure that the effect of the replicated method on replicated state stays the same at all object replicas, even if these return values are different at different participants.

An alternative to this direct interaction would be a strictly indirect indirection with local objects. The indirect approach has the advantage that shared objects do not need to maintain references to local objects outside the shared space; instead, this task is taken over by the platform infrastructure. Here, local objects register themselves for events on shared objects and shared objects signal events to the platform infrastructure. We illustrate such an indirect interaction in comparison with direct interaction in Figure 4.4. Croquet's TeaTime implementation actually provides such an indirect mechanism, which can also be used in Archipelago: Local objects can register themselves as subscribers of an event of a certain type at a shared object. Shared objects can then signal these events with a number of arguments and the platform infrastructure relays these to all subscribers.

A disadvantage of the indirect approach, however, is its limited flexibility. We have observed that, for the implementation of advanced coordination mechanisms in the shared space, the ability to interact with local objects

directly is more convenient, because such mechanisms often implement custom event subscription services, which, for example, require advanced matching strategies. Therefore, Archipelago/S allows shared objects to use both direct and indirect communication for interaction with local objects.

Parameter Passing

Another open issue is that of the way that parameters are passed into the shared space. Our current implementation based only on passing local objects by value may not suffice. For example, in some cases, such as the addition of objects to a shared list, passing by migration may be closer to the semantics of the message sent. Therefore, we currently also investigate the support of passing by migration and ways to define the parameter passing policy per message or method. The latter, for example, could be achieved by annotating method implementations with parameter passing policies for arguments and return values. We also plan to evaluate different settings for the default parameter passing behaviours.

4.5.2 Limitations of the Implementation

Archipelago/S is an implementation of the Archipelago platform design and, therefore, shares the limitations of the platform design itself, such as its restriction to a single programming language (here Squeak/Smalltalk).

Our current implementation also requires the source code of all shared objects to be manually deployed to all participants. This limitation could be solved by adding support for mobile code to Archipelago: A participant could deploy shared objects together with the source code that defines their functionality, which can then be dynamically installed in the runtime systems of other participants. An interesting idea could also be to support multiple different versions of the same class in a single VM image, but this would require significant changes to the Squeak VM and compiler.

Furthermore, the message router of the TeaTime architecture constitutes a single point of failure in the Archipelago platform implementation, which counteracts the argument of replicating shared objects for reliability reasons. However, we believe that a fully replicated implementation with a distributed two-phase commit as originally intended by the Croquet project and discussed in Reed's original work on the topic [41] can replace the central message router. The TeaTime architecture also results in the

limitation that temporary network outages may result in message loss and require loading a complete replica of the island.

Our implementation of auto message makers trades convenience and synchronous replicated message sends for performance: Because of their nature, auto message makers have to wait for the execution of the replicated message to be completed before returning execution back to the sender. This can have a large impact if many replicated messages are sent one after another, because the auto message maker will wait after each single new message, contrary to the use of manual `#future` message sends. In this regard, we see possibilities for improvement in smart just-in-time compilation of the message sending via auto message makers. If the return value of such a message send is not used by the sender, it could be compiled as an asynchronous message send. Then, however, any future local message sends to shared objects will have to ensure that previously sent replicated messages by the same participant are executed before the local message is send, so that the synchronicity of replicated messages and their effects can be guaranteed.

Another implementation limitation is the passing of Smalltalk blocks in replicated messages to shared objects: As these blocks have to be serialized in order to be passed into the island, we can only support blocks that solely access static variable values. There is no easy way to solve this problem. One idea is to use a transparent replacement of the block with a proxy object that, on invocation of the block, relays the call to the original local version of the block at the participant that passed it into the island. However, this solution makes the replicated execution dependent on a single participant and, because of this, is suboptimal.

In Chapter 3, we outlined that the Archipelago platform should allow the development of shared objects just like local objects. While Archipelago/S comes very close to this—shared objects are implemented in the same programming language and later simply deployed into the shared space—, there are also some restrictions for the implementation of shared objects. For example, developers cannot use infinite loops with sleep-phases, such as a game loop, within replicated behaviour. Instead, they have to make use of TeaTime's `#future:` mechanism to send messages to themselves or other shared objects to be executed at some point in the future. Another example is that developers also have to be careful not to use multi-threading for replicated computation that may have side-effects on replicated state, because computational determinism cannot be guaranteed in this case.

Other limitations of our prototype implementation are related to its limited support for debugging. For example, dealing with uncaught runtime errors during the execution of replicated messages is difficult: Such an error is currently escalated in each replica, instead of redirecting and handling it at the sender of the replicated message that caused it.

4.6 Summary

In this chapter, we have introduced Archipelago/S, our prototype implementation of the Archipelago platform design. Based on the Islands and TeaTime technologies of the Croquet system, it replicates computation to implement shared object spaces inside the Squeak/Smalltalk runtime system. We described our modifications and extensions of Croquet's replication technologies, particularly boundary objects and auto message makers as mechanisms to encapsulate the distribution boundary within shared objects.

Furthermore, we discussed a few issues that represent topics of ongoing research, including issues resulting from making shared state and functionality freely accessible to local functionality and vice-versa, and issues concerning our choice for the parameter passing behaviour. We also discussed several limitations of the Archipelago/S prototype, such as the central message router, which presents a single point of failure in the otherwise failure-tolerant platform implementation.

5 Evaluation

In this chapter, we provide a preliminary evaluation of the Archipelago platform and our prototype implementation Archipelago/S. We show how Archipelago/S can be used to implement a generic tuple space called ArchSpaces (Section 5.1) and how this tuple space and other Archipelago concepts can then be used to implement an exemplary distributed application, the multi-player networked game SpaceFight (Section 5.2). Subsequently, we summarize and discuss what we observed and learned through the application of Archipelago/S for these purposes, compare the Archipelago platform with related work, and outline topics of future work (Section 5.3). Finally, we summarize the chapter (Section 5.4).

5.1 ArchSpaces: Tuple Spaces in Archipelago/S

As described in Section 2.2.3, tuple spaces are a coordination model that can be used for both indirect, decoupled communication and coordination, as well as sharing of simple structured data between parts of a distributed application. We argue that, because of this reason, tuple spaces are a good example for a generic coordination and data sharing mechanism that is well suited to demonstrate how Archipelago can be used to implement, customize, and extend such mechanisms.

We present ArchSpaces, a simple tuple space implementation for our Archipelago/S prototype. We built ArchSpaces as a general-purpose, reusable tuple space for Squeak (Section 5.1.1), which we then deployed to Archipelago/S (Section 5.1.2). We discuss our implementation of ArchSpaces on top of Archipelago/S and its limitations in Section 5.1.3.

5.1.1 Tuple Space Interface and Implementation

The basic ArchSpaces implementation is independent of the Archipelago/S platform and does not consider any of its object sharing or replication-specific aspects. It is a simple tuple space implementation for local use in

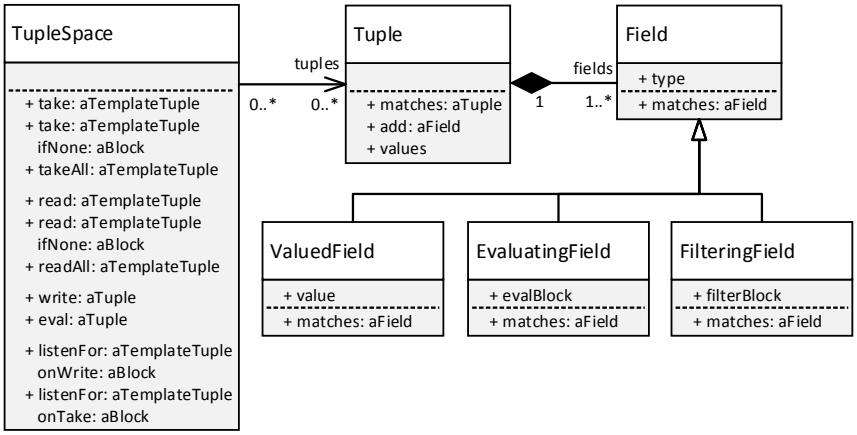

Figure 5.1: Class model of ArchSpaces (simplified).

a Squeak/Smalltalk image. This is possible, because we can simply deploy the object-oriented Squeak implementation of ArchSpaces to the shared object spaces of Archipelago/S with minor annotations and modifications, as described in the next subsection.

ArchSpaces implements the write, take, read, and eval operations of the Linda interface [19], as described in Section 2.2.3: `#write:` adds a new tuple by copying all its values, `#take:` removes a tuple matching a given tuple template, `#read:` is the non-destructive version of `#take:` and `#eval:` takes a tuple whose fields can be evaluated in a new process in the tuple space. We implement both blocking and non-blocking versions of the `take` and `read` operations: `#take:` and `#read:` are blocking, `#take:ifNone:` and `#read:ifNone:` are non-blocking and take a Block as second argument that is evaluated if no matching tuple is found. Additionally, we extend the interface with the operations `#readAll:` and `#takeAll:` to read or take all tuples matching a given template. We further add features for event-based interaction: Users can register event listeners for newly added or removed tuples matching a given template via the messages `#listenFor:onWrite:` and `#listenFor:onTake:`.

ArchSpaces is based on an object-oriented tuple space model, which is depicted in Figure 5.1. A tuple is represented as an object of the `Tuple` class, which maintains an ordered list of `Field` objects. The fields of tuples stored in the space are all of the `ValuedField` type and store a value of a certain type, tuples added by the `eval` operation can also include

`EvaluatingFields` storing a code block that is then evaluated to transform the field into a `ValuedField`. A tuple template is also represented as a `Tuple` object, whose fields can each be of three types: A `Field` filters for values of a specific type or can act as a wildcard value if its type is set to a `nil` value, a `ValuedField` filters for a specific value and a `FilteringField` allows filtering based on a custom filter represented as a code block.

```
1  | space tuple template taken |
2  "Create new Tuple Space instance."
3  space := TupleSpace new.
4
5  "Write a test tuple to the space."
6  tuple := Tuple new
7              add: (#name asValuedField);
8              add: ('test' asValuedField);
9              yourself.
10 space write: tuple.
11
12 "Read the tuple from the space."
13 template := Tuple new
14                add: (#name asValuedField);
15                add: (nil asField);
16                yourself.
17 taken := space take: template.
18
19 self assert: tuple equals: taken.
20
21 "Alternative tuple specification with bar symbol."
22 space write: (#name | 'test').
23 taken := space take: (#name | nil).
24 self assert: tuple equals: taken.
25
26 "Add and demonstrate write and take listeners."
27 space listenFor: (#name | nil) onWrite: [:tuple |
28    Transcript showln: 'written: ', tuple asString].
29 space listenFor: (#name | nil) onTake: [:tuple |
30    Transcript showln: 'taken:   ', tuple asString].
31
32 space write: (#name | 'test').   "Output: 'written: (#name | 'test')'"
33 space take:  (#name | 'test').   "Output: 'taken:   (#name | 'test')'"
```

Listing 5.1: Interaction with an ArchSpaces tuple space.

Listing 5.1 demonstrates the interaction with an ArchSpaces tuple space. We show how tuple objects can be created, added, and removed from the tuple space (lines 6–17). Our implementation also allows a more convenient way to specify tuples with the help of the vertical bar (pipe) symbol, as shown in lines 22 and 23. When tuples are specified in this way, the system automatically transforms classes or `nil` values into tuple fields of the type `Field`, block values into tuple fields of the type `EvaluatingField`

and other values into tuple fields of the type `ValuedField`. If other field types are required, values can be converted to these types by sending one of the messages `#asField`, `#asValuedField`, `#asEvaluatingField`, or `#asFilteringField` to it.

The general-purpose implementation of ArchSpaces uses a simple ordered list to store all tuples contained in the space. When a user asks to retrieve a tuple matching a certain template, the implementation performs a linear scan through this ordered list of tuples. The advantage of this solution is that the order of tuples is preserved; however, disadvantages include its lookup performance and scalability.

The ArchSpaces implementation presented so far provides the typical coordination, synchronization, and data sharing mechanisms of a generic tuple space and also enables event-based programming and indirect communication in a publish-subscribe-like fashion through tuple write and take listeners. Our implementation is aimed towards extensibility. We expect users of ArchSpaces to customize the tuple space implementation to their needs, for example, by extending it with additional features such as transactions, leasing, garbage collection, and persistence, or by adding performance improvements or a custom tuple space interface. The object-oriented design of ArchSpaces decouples the different tuple space implementation aspects and allows for extensibility through object composition and subclassing, aided, for example, by template and hook methods in the tuple addition, lookup, and removal functionality.

5.1.2 Deploying the Tuple Space to Archipelago

To deploy ArchSpaces onto Archipelago/S, we specified the distribution boundary of the tuple space by annotating its methods and extended ArchSpaces with services that allow attaching replicated functionality to tuple space events.

Specifying Local and Remote Data and Execution

For the issues of reusability and separation of concerns, the decision which tuple space functionality should cross the distribution boundary should be encapsulated from the users of ArchSpaces. Because of this, we converted `TupleSpace` instances to become boundary objects for the deployment of ArchSpaces to Archipelago. Users of the ArchSpaces implementation can

then obtain a reference to an auto message maker proxy for the tuple space and interact with it in the same way as with a local tuple space object.

For the implementation of the tuple space boundary object, we let the `TupleSpace` class inherit from `BoundaryObject` and specified which operations on a tuple space should be performed locally. Listing 5.2 shows examples for this labelling.

```
1  TupleSpace>>write: aValueTuple
2      self basicAddTuple: aValueTuple.
3      self fireWriteListenersFor: aValueTuple
4
5  TupleSpace>>take: aTemplateTuple
6      | tuple |
7      tuple := self basicRemoveTuple: aTemplateTuple.
8      self fireTakeListenersFor: tuple
9      ↑ tuple
10
11 TupleSpace>>read: aTemplateTuple
12     <executeLocally>
13     ↑ self findTuple: aTemplateTuple
14
15 TupleSpace>>listeners
16     <storeLocally>
17     ↑ listeners
18
19 TupleSpace>>listenFor: aTemplateTuple onWrite: aBlock
20     <executeLocally>
21     ↑ self addListener: aTemplateTuple onWrite: aBlock
22
23 TupleSpace>>listenFor: aTemplateTuple onTake: aBlock
24     <executeLocally>
25     ↑ self addListener: aTemplateTuple onTake: aBlock
```

Listing 5.2: Examples for labelling of TupleSpace operations: read is executed locally and write and take are replicated messages, but fire locally stored listeners.

The read operations can be performed without replicating their execution, as they have no side-effects on the state of the tuple space. There is no harm in executing them out of order with modifications of other participants to the tuple space, since their behaviour specification only necessitates that they access a consistent state of the tuple space. In order to synchronize changes to tuples, participants use the take operations instead. Therefore, we annotated the `#read:`, `#read:ifNone:`, and `#readAll:` methods of the `TupleSpace` class to be executed locally.

Furthermore, objects outside the island should be able to attach listeners to the tuple space. Hence, the event listener functionality also belongs to the local side of the object. Therefore, we labelled the instance variable

holding the list of event listeners as local state, and the methods that add or remove event listeners as local functionality. All other operations on the tuple space are replicated. When a write or take operation is executed, it also accesses the local list of event listeners at each participant and sends event notifications to matching listeners.

We already discussed in Section 4.4 that we extended the TeaTime technology with support for blocking replicated operations on shared objects. This allows us to use the blocking `#take:` message, which has to be replicated as it has side-effects on the list of tuples in the space. However, to be compatible with our solution for blocking operations, we had to modify ArchSpaces to use our custom TSemaphore class to implement the blocking.

The tuples in the space are data entities that, when obtained from outside the island, should be copied instead of passed by reference. The reason for this is that the tuple space operations themselves have a copying property: When a tuple is added to the space, the tuple and its values are copied into the space; and when it is retrieved through read or take operations, it is copied out of the space, too. Since, therefore, later accesses to tuple values do not need any replication, the overhead of sending and replicating messages to them into the island should be avoided. Hence, we declared the parameter passing policy for `Tuple` objects to be passing by deep copy. We chose passing by deep copy instead of passing by copy or clone, because all values within the tuple should be copied as well when the tuple is passed out of the island.

However, there are two limitations of this deployment as described so far: First, listeners cannot be used by functionality that resides inside the island, but only by objects residing outside of it. The reason for this is that listeners outside the island have to be executed asynchronously, so that they cannot influence the replicated execution, but functionality inside the island has to be deterministic and cannot use unordered asynchronicity. Second, code blocks cannot easily be serialized as arguments to replicated messages, as described in Section 4.5.2: This is only possible for code blocks that exclusively access static values. Because of this, evaluating tuples and custom filtering fields are very limited in functionality if the tuple space is accessed from outside the island.

Archipelago-Specific Extensions to the Interface

To overcome part of these limitations, we further extended ArchSpaces with support for replicated services. A service is a special kind of boundary object. Users of ArchSpaces can implement application-specific functionality in the shared space as subclasses of the `Service` class and use call-back mechanisms for tuple space events provided through it. A service can also replace an evaluating tuple, because, as a shared active object, it can exhibit behaviour similar to a process evaluating a tuple.

Figure 5.2 shows the extension of the ArchSpaces class model with the service functionality. We implement services through a set of jobs, which are registered at the service. A job can be registered for execution when a specific tuple event occurs or for periodic execution with a specific time interval. Correspondingly, a job is an instance of either the `TupleEventJob` or `TimedJob` class. It is registered by a custom `Service` subclass, such as the exemplary `MyService` class shown in the figure. Therefore, the custom service class overrides the abstract method `registerJobs`, which is executed when the service is registered in a new tuple space, and uses the hook messages `#execute:onWriteOf:`, `#execute:onTakeOf:` or `#execute:every:` to register new jobs. Each of these messages takes as first argument a message selector that is sent to the service class when a matching tuple event occurs or after the specified time period has passed. Services are registered at the tuple space via the replicated message `#registerService:`, either by replicated or local functionality. An interesting point to note here is that this `#registerService:` message, when sent from outside the island, copies the given service object into the island—a better way to match its intention would be to move the service object into the island by migrating it there. Therefore, this message would be a candidate for a passing by migration strategy, as discussed in Section 4.5.1.

5.1.3 Discussion of ArchSpaces

ArchSpaces provides a tuple space implementation with a simple Linda-style interface that we extended with event functionality. We deployed this implementation to Archipelago/S by specifying and encapsulating the distribution boundary and adding a service mechanism for replicated event listeners. This allows ArchSpaces to be used for indirect communication, coordination and simple data sharing between parts of a distributed

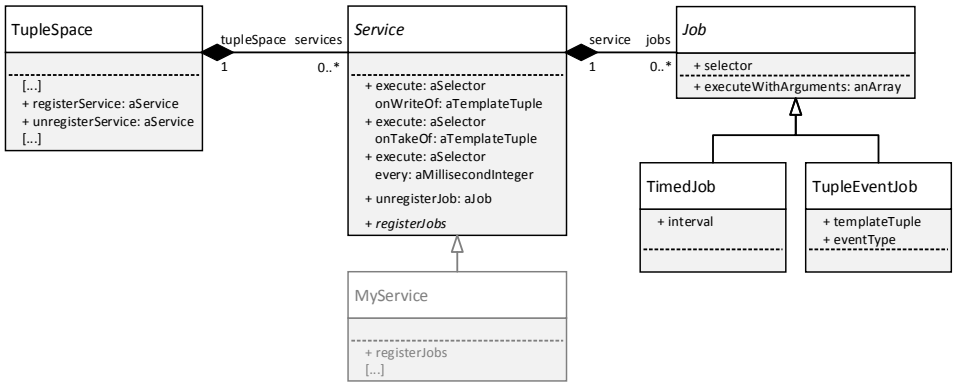

Figure 5.2: Extension of ArchSpaces with replicated services and jobs (simplified).

application, and also enables its use for event-based programming in this context.

The Archipelago/S concepts allowed us to deploy the tuple space in a shared and replicated island and to encapsulate the distribution boundary within the tuple space's implementation: The different parts of a distributed application can access the tuple space as if it was a local object. However, this is only partly true for the interaction of other shared objects with the tuple space, as discussed below in "Limitations".

The biggest advantage of the ArchSpaces implementation and its deployment to Archipelago/S is its easy extensibility. ArchSpaces can easily be extended with other features, such as transactions, leasing, or garbage collection. Considering the example from Section 2.4.1, where we discussed how to extend the SQLSpaces implementation with support for tuple dependencies, we can note that a similar extension of ArchSpaces can be performed very easily: It only requires a change in the object-oriented model of the tuple space to add the dependency association between tuples and the extension of the tuple removal logic with functionality that also removes all dependents of a tuple. The former can be accomplished by adding an instance variable and accessor messages for it to the `Tuple` class, the latter by overriding and extending the implementation of the internal method `#basicRemoveTuple:` of the `TupleSpace` class, as shown in Listing 5.3. In consequence, this extension only requires small changes of object implementations in the application's programming language. No proxy objects or serialization code has to be adapted, which is necessary for the extension of SQLSpaces.

```
1  DependencyTuple>>dependentTuples
2      ↑ dependentTuples ifNil: [dependentTuples := OrderedCollection new]
3
4  DependencyTupleSpace>>basicRemoveTuple: aTemplateTuple
5      tuple := super basicRemoveTuple: aTemplateTuple.
6      tuple dependentTuples do: [:dep | self take: dep].
7      ↑ tuple
```

Listing 5.3: Extension of ArchSpaces with tuple depedencies: We add a list of dependent tuples to Tuple objects and remove dependent tuples when their owner tuple is removed. We implement our extensions as subclasses of the Tuple and TupleSpace classes.

Archipelago/S also allows the combination of ArchSpaces with other shared data structures or coordination models deployed to the same island. We will discuss an example for such a combination with another data structure in Section 5.2.

Experiences Regarding Open Questions

During the deployment of ArchSpaces to Archipelago/S, we have made a few observations regarding the open questions of the Archipelago/S implementation discussed in Section 4.5.1.

Crossing the Distribution Boundary. Our choice to provide a mechanism for dividing a shared object into local and shared sides made it possible for us to define parts of the tuple space functionality to be executed locally, without crossing the distribution boundary. This way, we could avoid costly replication efforts for certain operations, such as the read operation, and, therefore, improve their performance. Moving these operations before the distribution boundary is easy because of the simple annotation mechanism of the boundary objects: We can simply access the shared state from the local side of the object and can be sure that we access a consistent state. However, we need to manually ensure that local operations do not modify any shared state. We have also not yet measured the performance advantages of local operations in detail and leave this task as future work.

Interacting with Local Objects. For ArchSpaces, we chose to use the direct approach to interact with local objects from inside the shared space. This allowed us to implement ArchSpaces' event handling mechanism

based on custom tuple template matching strategies. It also allows us to provide our own, custom interface at the tuple space object to add or remove event handlers in the form of Smalltalk blocks. Either of these would not be as easily possible with Croquet's generic event mechanism: It does not provide for custom matching strategies or Smalltalk blocks as handlers. An alternative for this might be to provide a more customizable generic event mechanism, which may allow these concepts. We are leaving the investigation and evaluation of such an approach as future work, too.

Parameter Passing. Regarding the parameter passing policy of Archipelago/S, we have seen one use case for a passing by migration policy: Passing the argument to the `#registerService:` method by migration would be closer to the method's semantics. Otherwise, the tuple space operations have copying semantics: Writing and reading tuples from the tuple space duplicates objects to make sure that the tuples inside the tuple space are only modified through the tuple space operations and not by modifying tuple objects.

However, Archipelago/S' argument copying also results in the duplicate copying of tuple objects, when the tuple space is used from outside the tuple space. If a tuple is sent to the tuple space, for example as argument to the `#write:` message, the tuple is first copied to cross the island boundary and then by the tuple space semantics themselves. We cannot avoid the first copy, since it is necessary to preserve the image segment consistency. However, it might be possible to remove the second copy by the tuple space semantics: In the case described, it is not necessary, because the tuple object passed to the tuple space is not referenced anywhere outside the tuple space itself. On the other hand, this copy is necessary if the tuple space is accessed from within the island: In such a case, the tuple objects are not copied by Archipelago/S, since no island boundary is crossed. Therefore, avoiding this copy is only possible in certain contexts, which makes this very difficult to realize. One possibility could be to always proxy accesses to the tuple space (even within the island) with a special proxy that acts differently based on whether messages are sent to it from within or outside the island and ensures that the arguments to method calls are copied. We chose not to pursue this approach any further, because it is a very specific performance optimization with limited impact.

Limitations

As discussed in Section 4.5.2, Archipelago/S is restricted in the usage of Smalltalk blocks as or within arguments to replicated messages. We have observed this limitation when developing ArchSpaces, too: Evaluating or filtering tuple fields can only be used with the limitation that the blocks have to be serializable and can only access static values. The advantage of the Archipelago platform in this sense is that custom shared active objects can replace evaluating tuples.

However, we have seen an additional problem in the interaction of shared objects with an ArchSpaces tuple space: They cannot use the event listener functionality of the tuple space, because event listeners are executed asynchronously. We could overcome this problem by adding an additional event mechanism for shared objects, which we call services. This, however, required two separate implementations of the event mechanism for the local and shared side of the tuple space.

One of the goals of the Archipelago platform is to allow the development of and interaction with shared objects to be as close to that of/with common local objects. The problems described above show the limitations of our platform prototype regarding the realization of these goals. Developers of objects outside the island have to keep Smalltalk block serialization in mind when interacting with a replicated tuple space, and developers of shared objects have to use services for such an interaction. While the block serialization is very difficult to resolve—it may even be impossible—, we still plan to investigate alternative solutions that may allow the implementation of a tuple space event listener interface in the same way for both local and shared objects. One way to do this could be to supply a more customizable generic event handling mechanism through the Archipelago/S platform, which allows custom matching strategies and event handler specifications, and use this mechanism to implement ArchSpaces' event handling. In this case, the platform could make sure that event handlers added from inside the island are stored and treated differently from those added from outside the island.

5.2 SpaceFight: An Example Application

The Archipelago platform supports communication, coordination, and data sharing in distributed applications. In the previous section, we have

shown how it can be used to develop the ArchSpaces tuple space, as an example for an extensible coordination mechanism built with Archipelago/S. In this section, we develop an exemplary distributed application on top of Archipelago/S, the multi-player networked game `SpaceFight`. We show how ArchSpaces can be used, customized and extended for the development of this game and how Archipelago/S can further be used to combine ArchSpaces with a custom shared tree data structure and to deploy shared functionality specific to SpaceFight into Archipelago's shared object space.

We first outline the game concept (Section 5.2.1) and then sketch the architecture of the game implementation (Section 5.2.2). Subsequently, we describe the customization of ArchSpaces for use in the SpaceFight game (Section 5.2.3) and explain how we use the Archipelago concepts to implement functionality shared by all game participants and to combine ArchSpaces with a custom tree data structure (Section 5.2.4). Finally, we discuss our implementation of the SpaceFight game and its limitations (Section 5.2.5).

5.2.1 Introduction to SpaceFight

SpaceFight borrows from games such as SpaceWar or Asteroids, a screenshot is shown in Figure 5.3. Each player navigates a ship through an asteroid-filled region in space. A collision with an asteroid destroys a ship, however, the ships can destroy asteroids or other ships by firing missiles at them. Each player's goal is to win the game by outliving all other players' ships.

SpaceFight also allows each player to customize, extend, or otherwise modify parts of the game. For example, players can use custom controls to navigate their ships, customize the user-interface to their taste , replace the manual controls by autonomous strategies, or add weapon system upgrades .

The conceptual architecture of our game, therefore, is a distributed peer-to-peer application consisting of computationally different parts: The players constitute the distributed components of this application, which communicate with each other and share state and computation to coordinate the game's execution. To apply customizations or modifications, players can extend the implementation of their local game components.

As typical for distributed applications, the interaction of the different distributed components of the SpaceFight game requires some shared

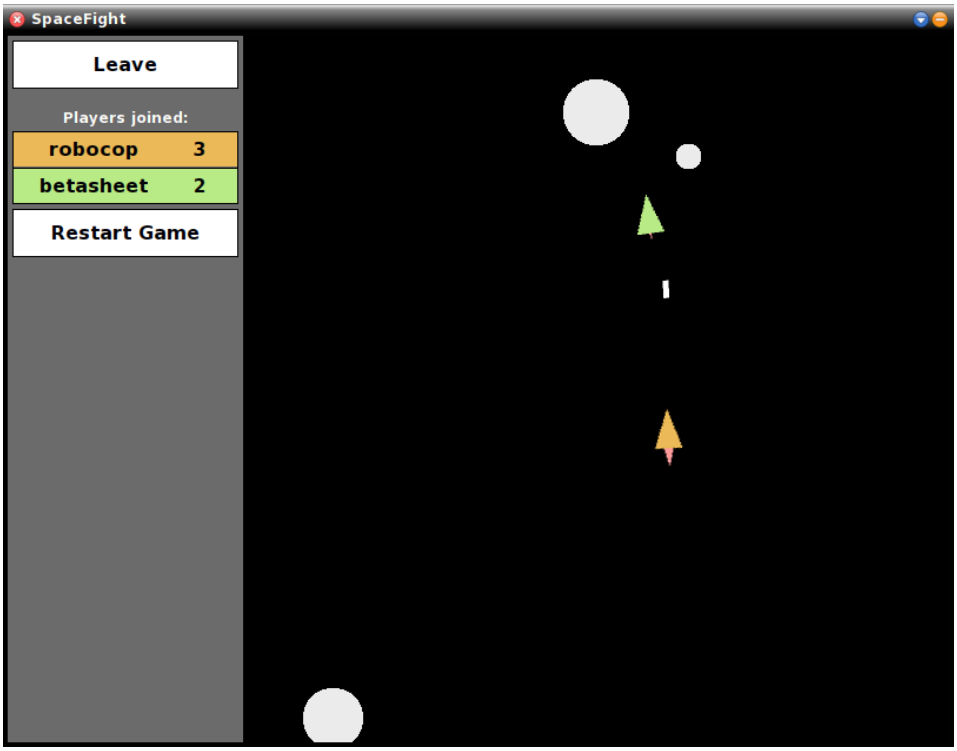

Figure 5.3: Screenshot of the SpaceFight game. We show the local game interface of the player *betasheet*. The player's green ship is heading towards two asteroids, and a second player, *robocop*, is shooting in his direction.

state and coordination. Examples for shared state include the position and movement of game entities, such as ships, asteroids, and missiles, general game state, such as the game time, participating players, and their remaining lives, as well as game statistics. The components also have to solve a variety of coordination tasks, such as recognising and reacting to game state changes, changes in movement vectors of entities, or entity destructions.

To solve most of these coordination and data sharing tasks in SpaceFight, we use our ArchSpaces tuple space technology that we described in the previous section and extend and customize it for use in the SpaceFight game.

As we discussed in Chapter 2, a tuple space is well suited for simple, generic coordination and data sharing tasks, but other mechanisms may be

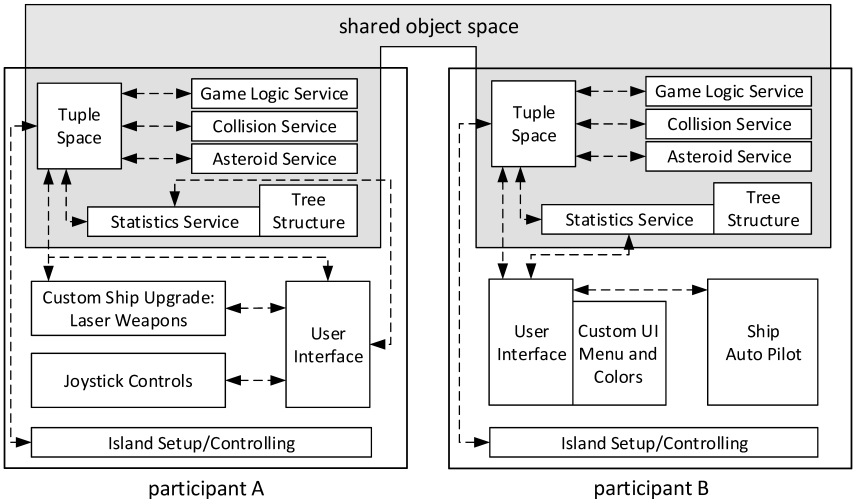

Figure 5.4: SpaceFight game architecture. Here, we show the shared and local components in a setup with two participants and their interactions.

better suited for more specialized or more complex tasks. In our game, the participants share result statistics between each other. These statistics are sorted key-value pairs, but their lifetime surpasses a single game instance. Therefore, all participants persist the statistics locally and contribute them to the shared statistics pool while avoiding to add duplicate result entries. Furthermore, the number of entries is comparatively high and the components require predecessor and successor queries on the entries in order to show other results around a specific player's result. Because of these properties, we do not use the tuple space to store the shared statistics, but instead deploy a custom tree-based data structure that supports predecessor and successor operations, as well as duplicate detection.

Furthermore, there are certain computations within the game execution whose effects have to be consistent among all participants, such as the detection of collisions, changing game states, and spawning new asteroids. In our peer-to-peer architecture, the responsibility for these computations is shared amongst all participants. We use Archipelago's concepts to implement these computations as application-specific functionality inside the shared object space.

5.2.2 Architecture of the Game

The game architecture is illustrated in Figure 5.4. We use a single island for all shared state and functionality between the participants of the game. Components inside the island include the extended tuple space and services for game state changes, asteroid creation, collision detection, and statistics storage. As described in the previous section, we use the tuple space, for example, to store values of the game state and position vectors and attributes of game entities, but also for coordination tasks: In this context, it functions as a mediator for a publish-subscribe event pattern to decouple the different services from each other. The game logic service, for example, listens for new tuples describing entity collisions, which are written to the space by the collision detection service. The statistics service also uses an additional shared tree data structure for storing and retrieving statistics data.

The functionality that may not be the same for each player resides outside the island: the user interface, individual customizations, such as autonomous piloting, custom ship designs, or weapon upgrades, and the control logic that sets up or joins the island. The game currently allows any player to be the message router of the island, however, this player then becomes a single point of failure. Here, the central router of the Croquet architecture becomes a major limitation of the architecture; we discussed this limitation in detail in Section 4.5.2.

5.2.3 An Application-Specific Tuple Space

ArchSpaces provides a very simple, generic tuple space implementation. For use in SpaceFight, a customized version of this tuple space implementation is better suited, as there are certain additional requirements on the tuple space implementation for use in the SpaceFight game.

For our game, the look-up performance of the tuple space is important: Read or take operations are executed frequently, for example, during the detection of collisions, and need to be fast in order to achieve good responsiveness and frame rates of the game. The Archipelago platform allows us to easily extend the ArchSpaces implementation to fit these needs, because the tuple space is implemented as an object-oriented structure in the application runtime. We exploit this to replace the single list in the ArchSpaces implementation that stores the tuples in a space by hashing the

tuples into multiple lists based on their number of fields. This reduces the length of the lists and, with that, the lookup times necessary to find a tuple matching a given template. We implemented this extension by subclassing ArchSpaces' `TupleSpace` class and making changes to the internal tuple space functionality that adds tuples to the list and performs the search for matching tuples within the list, as shown in Listing 5.4. ArchSpaces would also allow us to develop even more advanced extensions to improve the access/search time, such as creating indexes for tuple values.

```
1  SFTupleSpace>>lists
2      "dictionary holding the lists that store the tuples."
3      ↑ lists ifNil: [lists := Dictionary new]
4
5  SFTupleSpace>>listFor: aTuple
6      "return the list that the given tuple should be stored into."
7      ↑ self lists at: aTuple size ifAbsentPut: [OrderedCollection new]
8
9  SFTupleSpace>>basicAddTuple: aTuple
10     "override to store into different lists based on tuple size."
11     ↑ (self listFor: aTuple)
12         add: aTuple
13
14 SFTupleSpace>>basicRemoveTuple: aTuple
15     "override to remove from different lists based on tuple size."
16     ↑ (self listFor: aTuple)
17         remove: aTuple
18
19 SFTupleSpace>>basicFindTuple: aTuple ifNone: aNoneBlock
20     "override to search within different lists based on tuple size."
21     ↑ (self listFor: aTuple)
22         detect: [:each | aTuple matches: each]
23         ifNone: aNoneBlock
```

Listing 5.4: Extending the TupleSpace with multiple tuple lists (excerpt).

We can also use Archipelago's concepts to solve a problem inherent to generic tuple spaces, which we described briefly in Section 2.2.3. As a result of using the generic tuple space interface for coordination and sharing data, the users of the tuple space are data-coupled, that is, they all need to know the exact format of the tuples. This prevents encapsulating parts of a tuple format from users and can lead to issues when the tuple format evolves.

To illustrate this problem, consider the following example: The collision detection service may needs to distinguish between different types of entities. Therefore, for each entity, it reads tuples of the format `(#entityType | [entityId] | [entityType])` from the space that associate entity ids

with their respective types. Later in the development process, the tuple format is changed to also include a reference to the player that owns the entity: `(#entity | [entityId] | [entityType] | [playerId])`. For the collision detection, this new field is not of interest; however, it needs to know about its existence to find the correct tuple in the space. Otherwise, the template used for lookup will not match the correct tuples in the space.

Therefore, tuple spaces are therefore often accessed through customized proxy objects that allow developers to specify application-specific operations on the tuple space, such as a method `#entityTypeFor:`, which looks up the tuple that stores the entity type for a given entity identifier and returns the type. This way, the tuple format is encapsulated in the custom proxy object and hidden from the user, which, for example, is more flexible towards evolving tuple formats.

For the SpaceFight game, we extend our custom tuple space class with such an application-specific interface. The Archipelago architecture further allows us to execute parts of this functionality behind the distribution boundary, grouping multiple operations on the tuple space into a single replicated operation, which can reduce the underlying network communication. We show an excerpt of this interface in Listing 5.5.

5.2.4 Application-Specific Functionality in the Island

As described in Section 5.2.1, our game contains functionality that all participants share the responsibility for and whose effects have to be consistent amongst all participants. Archipelago/S allows us to transfer the responsibility of executing this functionality to the platform, by making it shared functionality deployed to the island. We have implemented the shared functionality of our game as services of the tuple space, to enable it to access the tuple space from within the island. These services detect game state conditions and react to game state changes (game logic service), randomly spawn asteroids on the map (asteroids service) and perform the task of detecting collisions (collision service).

We further combine the tuple space with a custom shared tree structure for storing SpaceFight statistics, as explained in Section 5.2.1. This structure is maintained by and accessed via the statistics service. Participants export locally stored statistics into this structure when they join a new game and save a copy of the shared statistics after each game round, and the user-interface can use the structure to display results. Our tree

```
SFTupleSpace>>entityTypeFor: anEntityId
    | tuple |
    "return the entity type of the entity with the given id.
    this translates to a read operation and can be executed locally"
    <executeLocally>

    tuple := self read: (#entity | anEntityId | nil | nil).
    ↑ tuple third

SFTupleSpace>>writeEntityId: id type: type playerId: playerId
    "write a new entity type and player id for the entity with the given id.
    this translates to a write operation -> execute replicated"

    self write: (#entity | id | type | playerId asAValuedField)

SFTupleSpace>>nextEntityId
    | tuple id |
    "return the next free entity id.
    we use a take and write operation, which need to be replicated.
    therefore, it makes sense to execute this message replicated
    to group the take and write into a single replicated message."

    tuple := self take: (#nextEntityId | Integer).
    id := tuple second.
    self write: (#nextEntityId | (id + 1)).
    ↑ id
```

Listing 5.5: Excerpt of our application-specific interface to the TupleSpace. We show how custom local and replicated messages can be defined, and how multiple replicated messages can be grouped into a single custom one.

structure implementation is based on a BTree implementation for Squeak[1], which we extended with range searching functionality and deployed to Archipelago/S by annotating local and replicated operations, similar to the annotations for the deployment of ArchSpaces. Here, the advantages of the Archipelago platform payed off: We could use an existing implementation of a structure written in the same programming language, extend it to our needs, and simply deploy it with little effort to the shared object space. An excerpt of the implementation of the statistics service is shown in Listing 5.6.

5.2.5 Discussion of SpaceFight

For the development of SpaceFight, we made use of the Archipelago concepts for two main purposes: First, we deployed an extended and customized implementation of the ArchSpaces tuple space and combined it

[1] BTree implementation from http://www.squeaksource.com/BTree.html (2014/09/16)

```
1  StatisticsService>>scoreTree
2      ↑ scoreTree ifNil: [scoreTree := BTree new]
3
4  StatisticsService>>addScore: aScore
5      "add a new score to the tree, while avoiding duplicates."
6      self scoreTree at: aScore ifAbsentPut: aScore
7
8  StatisticsService>>addAllScores: aCollectionOfScores
9      "add many scores to the tree, while avoiding duplicates.
10     used by new participants to contribute all of their locally
11     stored scores to the shared pool of scores."
12     aCollectionOfScores do: [:each | self addScore: each]
13
14 StatisticsService>>find: anInteger scoresAround: aScore
15     "find a number of scores that surround a given score via a
16     range query to the tree. used by participants to obtain a
17     list of scores around a player's score to show after a game."
18     <executeLocally>   "non-modifying operation"
19     ↑ self scoreTree find: anInteger keysAround: aScore
20
21 StatisticsService>>allScores
22     "return all scores in the shared pool. used by participants
23     to copy the shared statistics pool into their local storage
24     after each game."
25     <executeLocally>   "non-modifying operation"
26     ↑ self scoreTree values
```

Listing 5.6: Excerpt of the StatisticsService implementation. All operations but the lookup operations `#find:scoresAround:` and `#allScores` are replicated.

with a custom shared tree data structure to solve many communication, coordination, and data sharing tasks in SpaceFight. Second, we moved parts of the functionality specific to SpaceFight into the shared space.

Using a tuple space that was developed according to our Archipelago concepts, that is, in the same programming language and runtime environment, allowed us to easily extend its functionality by following object-oriented principles: We simply subclassed the tuple space implementation and overrode specific methods to replace the internal storage of tuples with a more efficient one. The changes were confined to a single class and could be performed in the same development environment and programming language.

Furthermore, these properties enabled us to replace the generic tuple space interface with a more specialized one, which we used to encapsulate tuple formats from users, thereby decreasing data coupling between different parts of the game. We also used Archipelago's properties to combine the tuple space with a separate custom tree data structure. We were able to use the same platform and infrastructure for this custom structure by

deploying it into the same replicated island. Archipelago/S also allowed us to reuse and extend an existing binary tree implementation for this use case, which was developed as if it was a local structure.

Archipelago's concepts also allowed us to move parts of SpaceFight's application-specific functionality into the shared space, thereby giving the responsibility for this functionality to the platform rather than an individual participant. This is possible, because objects in the shared space of the Archipelago platform are independent of the participants, and is particularly useful for the development of applications such as SpaceFight, which make use of a peer-to-peer architecture where participants can freely join or leave the distributed application setup. We argue that this can simplify the architecture and implementation of applications such as SpaceFight.

For SpaceFight, this allowed us to achieve consistent collision detection, asteroid spawns and game state changes across all participants without the need for complicated coordination strategies. By implementing this functionality as shared objects inside the replicated island of the Archipelago/S prototype, the only coordination necessary for ensuring the consistency of the effects of this functionality lies in coordinating which participant deploys the objects into the space. In SpaceFight, this is simply done by the participant that creates a new game instance. Afterwards, the consistency of the functionality's effects is guaranteed by the platform implementation. If we had not implemented this functionality inside the shared space, the participants themselves would have had to execute it locally and coordinate the consistency of its effects amongst themselves. For example, complicated strategies would have had to be put into place to decide which participant had the final say on whether a collision did or did not occur.

Experiences Regarding Open Questions

Crossing the Distribution Boundary. Our boundary objects made it easy to deploy both the tuple space and the binary tree structure. It also allowed their simple extension by containing the necessary changes within the objects themselves: It was not necessary to extend any proxy classes or serialization code. However, extending a boundary object is a little more difficult than extending a common object—The developers extending it need to have a good understanding of the internals of the object, so that they can make sure that no inconsistencies can arise in the shared state

of the object because of the changes they make. The extension of Archipelago/S with automated mechanisms to check for any such violations, as discussed in Section 4.5.1, might aid these tasks by providing more confidence in the validity of the changes.

Interacting with Local Objects. Besides the event handling mechanism for the tuple space, we have not found any use cases for the interaction with local objects—whether direct or indirect. This fact strengthens the case for a more customizable generic event handling mechanism, since this seems to be the main use case for interaction with local objects. Therefore, investigating and evaluating such a mechanism remains a highly prioritized goal of future work.

Parameter Passing. We have not seen any other use case for passing by migration behaviour during the implementation of our game. This is due mostly to the fact that the operations of our data structures inside the space (tuple space and tree structure) have copying semantics. However, we have noticed that some methods have intrinsic parameter passing behaviour that does not depend on the type of their arguments or return values: Our custom tuple space operations follow the semantics of the underlying tuple space operations. For example, our new message `#entityTypeFor:` follows the semantics of the read operation and, therefore, has copying semantics. However, the type returned by such a new message is user-defined and may not have copying semantics (but instead pass-by-reference semantics). Therefore, we believe that this would be a good use case for defining method- or message-specific parameter and return value passing policies, as we described in Section 4.5.1.

Performance

A typical issue for platforms such as Archipelago is their performance. Archipelago has to ensure the consistency of shared objects, a task which can involve large overheads because of its generic nature. Our current prototype implementation of Archipelago is not highly optimized for performance aspects, and, therefore, we refrained from a comparative performance evaluation.

Instead, we chose to develop the SpaceFight game as an exemplary application which demonstrates informally that our prototype implementation

can fulfill the performance requirements of a highly responsive network game. We tested this game with two to five participants in a local network setup and could achieve a fast responsiveness on each player without noticeable lag in user interactions, which could otherwise be caused by the replication of these interactions. Using a collision detection rate of 100 invocations per second, we achieved adaptive frame rates around 60 frames per second[2].

A general performance issue in our current implementation of Archipelago/S is the occurrence of system interruptions at a participant when a snapshot of its island replica is created. Furthermore, and as discussed in more detail in Section 4.5.2, our choice to execute replicated messages synchronously to the local execution that they were initiated from can incur performance loss compared to an asynchronous execution.

A Note on Tuple Spaces as Main Coordination Medium

In SpaceFight, we use a tuple space as main coordination medium that decouples the different participants. We have found that tuple spaces provide a good abstraction for the typical coordination tasks and are especially convenient when used with an event-based mechanism. This way, we were able to implement modular services that are highly decoupled from each other. One example for this is the decoupling of the collision and game logic services: The collision service regularly obtains current entity positions from the tuple space and writes tuples back into the space that denote a collision event. The game logic service listens for these events and executes collision handling logic when they occur. This way, the tasks of detecting and handling collisions are well separated and decoupled, as is illustrated in Listing 5.7.

By extending the tuple space and its services with an application-specific interface, we could further reduce the data coupling of the participants and encapsulate tuple format knowledge.

Furthermore, we found that blocking tuple space operations are essential for solving synchronization problems, for example, when updating a tuple in the space. We first followed an approach for storing and updating entity movement vectors in the tuple space, where we read the old movement vector from the tuple in the tuple space, inserted a new tuple with an

[2] measured on Intel Core i7-3612QM 2.10 GHz (4 cores, hyperthreading), 8 GB RAM, connected to 100 Mbit/s local ethernet

```
1  CollisionService>>detectCollisionsAt: aTimeMillisInteger
2      "detect collisions based on entity positions at the given time."
3      entityVectors := self entityVectorsAt: aTimeMillisInteger
4      (self entityVectorPairsFrom: entityVectors) do: [:vector1 :vector2 |
5          (vector1 collidesWith: vector2) ifTrue: [
6              self handleCollision: vector1 entityId
7                   with: vector2 entityId]]
8
9  CollisionService>>handleCollision: anEntityId with: anotherEntityId
10     "signal a collision between the two entities."
11     self tupleSpace signalCollision: anEntityId with: anotherEntityId
12
13 GameLogicService>>registerJobs
14     "register a job to handle entity collisions"
15     self executeOnEntityCollision: #collisionOf:with:
16
17 GameLogicService>>collisionOf: anEntityId with: anotherEntityId
18     "handle a collision between two entities."
19     entityType := self tupleSpace entityTypeFor: anEntityId.
20     anotherEntityType := self tupleSpace entityTypeFor: anotherEntityId.
21     self handleCollisionOf: entityType id: anEntityId
22          with: anotherEntityType id: anotherEntityId
23
24 SFTupleSpace>>signalCollision: anEntityId with: anotherEntityId
25     | tuple |
26     "signal a collision between the two entities."
27     tuple := #entityCollision | anEntityId | anotherEntityId.
28     self write: tuple;
29          take: tuple
30
31 SFService>>executeOnEntityCollision: aSelectorSymbol
32     "send the selector when a collision occurs.
33     we evaluate the tuple value before sending the selector
34     to support selectors with two parameters."
35     self execute: aSelectorSymbol
36          onWriteOf: (#entityCollision | nil | nil)
37          evaluateTupleValue: [:tuple | Array with: tuple second
38                                           with: tuple third]
```

Listing 5.7: Decoupling of the game logic and collision services via tuple space events. We use a custom interface for the tuple events, implemented in SpaceFight's extensions of the TupleSpace and Service classes.

updated movement vector, and then deleted the old tuple. This ensured that there was always a movement vector tuple existent in the space, and components interacting with the tuple space, such as the collision detection, used non-blocking operations to execute their logic. However, we found that this could lead to conflicting situations where a movement vector was changed by a component during its update by another component, resulting in consistent, but unintended state in the tuple space. We could solve this problem easily by using blocking operations for the tuple update instead. This also illustrates the need for supporting blocking operations on shared objects in the Archipelago/S implementation, which we described in Section 4.4.

Limitations

While we found that a custom, application-specific interface for the tuple space helped us resolve data dependency issues between parts of the game, our separate implementation of services and event listeners often required us to implement custom event registration messages once for use by services and another time for use by listeners. For example, for a tuple write event for the game state, we both implemented a message `#executeOnGameStateWrite:` for use by services, as well as a message `#onGameStateWrite:` for use by external event listeners. Here, the ArchSpaces' separate internal and external event mechanisms lead to duplicate work during its extension for SpaceFight.

Other limitations we observed include the topics discussed above under "Experiences Regarding Open Questions": Archipelago/S currently does not allow parameter passing behaviour to be defined per method or message, which would have been useful for our custom tuple space interface, and application developers may need additional insights into the implementation of a boundary object if they want to extend it in a safe way.

5.3 Discussion of our Observations

Our implementations of the ArchSpaces tuple space and the custom binary tree structure for the SpaceFight game demonstrated how Archipelago/S can be used to implement coordination models and shared data structures. We first developed these structures as local object structures and were then

able to deploy them to Archipelago/S' shared object space (the island) with only minor modifications: We converted the respective main classes of the two structures into boundary objects by laying down the configuration of the distribution boundary through simple method annotations and made some other small modifications, such as applying the TSemaphore classes to realize the blocking operations of the tuple space.

We argue that these modifications are quite contained and of small scope, since we were able to reuse the majority of the internal implementation of the local versions of the structures without any modifications. For example, the tuple matching logic of the tuple space was not affected by the deployment to Archipelago/S, and neither was the binary tree logic that determines the position of a certain value in the tree. However, we have also seen the limitations of the Archipelago/S prototype: The most major one being that the event handling functionality of the tuple space could only be made to work with event handlers outside the island, which forced us to implement a separate event handling mechanism for event handling inside the island in the form of the service functionality. Therefore, investigating alternative solutions to this remains a highly interesting topic of future work, as we discussed in Sections 5.1.3 and 5.2.5.

With the development of the SpaceFight game itself, we could demonstrate how ArchSpaces can be extended for application-specific purposes, how it can be combined with the binary tree structure inside the same island, and how pushing application-specific functionality inside the island can further simplify distributed application development. Extending and customizing ArchSpaces was made easy by using common object-oriented mechanisms, in particular, we used subclassing to apply performance improvements and add a custom interface to the tuple space. However, we also noticed that extending a boundary object requires detailed knowledge about the inner workings of the object to guarantee that the extensions cannot violate shared state consistency. Evaluating potential solutions to this also remains a topic of future work.

Archipelago/S as Replicated Implementation of the Platform Design

Through applying Archipelago/S in this context, we argue that we were able to show that our platform prototype can be successfully used to implement general-purpose coordination models and shared data structures, extend and customize them for application-specific purposes, and com-

bine them with other structures in the same platform. We showed this exemplarily for our tuple space and binary tree implementations and their adaptation for use in our distributed SpaceFight application.

Through Archipelago/S, we were able to evaluate the feasibility of our platform design concepts in a replicated setup. This replicated architecture automatically provides for high reliability, since all shared data and computation is replicated at each participant. Our implementation uses Croquet's Island and TeaTime technologies at its heart, which are the backbone in providing good performance by replicating computation. Our preliminary evaluation shows that the performance is good enough to support our use case of a real-time 2D game, but a detailed performance evaluation remains as future work. On the other hand, TeaTime's architecture relies on a central message router, which poses a single point of failure.

Replicating the island at all participants also means that the approach is not highly scalable in the size of the shared object space for two reasons: The time necessary to join an island increases linearly with its size, because it has to be transferred in full, and its maximum size is limited by the free space of each individual participant. Therefore, in this setup, scalability can only be achieved by keeping the size of each shared space to a minimum and instead combine multiple shared spaces to separate parts of the shared state and functionality that are relevant for different groups of participants.

In order to be able to further evaluate and compare the usability, performance, reliability, and scalability aspects of our prototype, we think that implementing the Archipelago concepts in different architectural setups, such as a distributed one or even a combined distributed and replicated one, and subsequent comparisons would be interesting topics of future research.

It would also be interesting to see if the open questions we raised in Section 4.5.1 are specific to Archipelago/S, or whether they transfer to other implementation approaches as well, which would suggest that they are intrinsic to the platform design. We know that the general topics, that is, the crossing of the distribution boundary, the interaction with local objects, and parameter passing policies, are platform design issues, which we also discussed in Section 3.3.3.

However, our boundary object solution and the associated issues are specific to Archipelago/S, since separating an object into shared and local sides was only easily possible because the object exists as copy in each participant's local runtime. We could imagine a similar solution for

other implementation approaches: For example, a distributed platform implementation could automatically generate proxy objects based on the boundary object specification. The proxies could then store local state, execute local functionality, and relay any accesses to the shared state to the real object located at a single participant in the distributed setup. In such a case, the issue that shared state consistency could be violated by local functionality does not exist (since only one copy of the shared state exists)—instead, we would trade this problem for a more complex platform implementation that has to realize generating the proxies. This hypothesis, however, has to be verified or proven wrong through implementation: In the process of which, we might discover other issues related to crossing the distribution boundary.

5.3.1 Comparison with Related Approaches

There are multiple other approaches for platforms that allow communication, coordination, and/or data sharing in distributed applications. In this subsection, we compare Archipelago with representatives of the distributed objects approach based on service-oriented architectures, technologies for sharing passive data structures, the Emerald language for mobile objects, the distributed shared objects concepts described by Homburg et al., and the GemStone object database management system.

Distributed Objects

One popular approach, exercised, for example, by CORBA [21], Java RMI [24], and Jini [53], is based on a distributed architecture and is commonly referred to as the *distributed objects* approach. It is based on remote procedure calls and service-oriented architectures, which we described in Sections 2.1.2 and 2.2.4. The service-oriented architecture of these platforms can be used directly as coordination mechanism itself, or can be used to implement other coordination mechanisms and shared data structures on top of the service-oriented architecture. Jini, for example, forms the basis for the JavaSpaces tuple space [52], a centralized service that exposes a tuple space interface to clients.

Thus, in the distributed objects approach, a participant can share objects by exporting them via a service registry component and other participants can use this service registry to look up and obtain references to shared

objects by means of service descriptions. As shared objects are local to a specific participant, they can access its local resources for their implementation. RMI and Jini further allow the developer of a shared object to implement a custom proxy object, a *smart proxy*, to encapsulate the distribution boundary, similar to the Half Object Plus Protocol pattern [12]. The smart proxy represents mobile code and can be dynamically installed on clients when the service is requested.

Contrary to Archipelago, shared objects in the distributed objects approach are local to and dependent on a particular participant; however, many concepts are comparable. Advertising an object as a service at the service registry is similar to deploying an object into the shared space of Archipelago—with the difference that deploying an object to Archipelago means giving away the responsibility for storing/distributing its state and executing its functionality into the platform; whereas in the distributed objects approach, the responsibility for this remains with the participant publishing the service.

Many of the responsibilities that the Archipelago platform design delegates to its implementation are also present in the distributed objects approach: The interaction with a service has to cross the distribution boundary, calling methods of a service requires parameter passing policies for arguments and return values, and services may also interact with each other.

Because of this, we can compare our implementation choices regarding these aspects for Archipelago/S with typical behaviours used in the distributed objects approach: Archipelago/S' boundary objects have a similar purpose as RMI/Jini smart proxies, but do not separate the proxy definition from the shared object. As we discussed before, this reduces the need for implementing separate caching mechanisms, but can also incur higher risks for shared state consistency violations. The parameter passing semantics of RMI and Jini are also similar to the ones we adopted for Archipelago/S - In these platforms, shared objects are usually passed by reference and other objects by value. In Archipelago/S, the same occurs for arguments to replicated methods, the return values of replicated methods exhibit slightly different parameter passing behaviour, since their type determines whether they are passed by reference or by value. Another difference between the approaches exists in the way the interface of shared objects is specified: While in CORBA, it has to be specified in a special interface definition language, and in RMI and Jini, it is specified as a sep-

arate Java interface, shared objects in Archipelago/S do not require any separate interface description; only the distribution boundary can optionally be specified through labelling with annotations. This allows users of Archipelago to easily share simple objects without having to add labels or interface descriptions.

Another interesting concept seen in Jini is the way that time is introduced into the allocation of shared resources: Jini services use a leasing mechanism to grant access to resources, which means that clients have to explicitly express interest into a resource as long as they wish to use it. This allows for a relatively flexible, use-driven garbage collection. An interesting topic for future work could be to investigate whether such a mechanism can be used to introduce garbage collection into Archipelago/S' islands.

Sharing Data Objects and Structures

Other approaches aim to support custom shared data structures through shared object spaces. The Orca language [6], for example, lets the application developer specify abstract shared object data types and their operations and interface with these objects as if they were local, while its run time system transparently assures their consistency and synchronization between the nodes. Orca does not follow the concept of object spaces where participants can deploy objects dynamically, but instead passes references to shared objects explicitly to other (new) processes in fork statements.

Orca only focuses on sharing data between application parts, while Archipelago also supports mechanisms to coordinate the application's execution. Contrary to Archipelago, shared objects in Orca are passive and their operations can only access the object's internal data. Because of this, Orca cannot be used to share application logic or to implement events on shared objects. Orca also hides the distribution boundary, making it completely transparent both to the user and developer of a shared object, while Archipelago and Archipelago/S allow its encapsulation and configuration.

The Virat [47] system is another approach for shared objects, which combines some ideas of tuple spaces, distributed shared memory, and object spaces. In Virat, passive data objects can be deployed into a shared object space, which is accessed via Linda-like write and read operations and associates objects with identifiers. This makes the distribution boundary explicit: Users have to write a modified object back into the space to make

changes visible to others. Similar to Orca, Virat does not intend shared objects to interact with each other. Additionally, references between shared objects are not intended either. It, therefore, also only targets sharing data, rather than supporting the implementation of coordination mechanisms or application logic in the shared object space.

Emerald: Mobile Objects

In the Emerald system [26], all objects of a distributed application are shared and mobile, that is, they can migrate their location freely and transparently. Objects can be passive or active; active ones have a process attached to them that can execute the object's behaviour. As in Orca, the distribution boundary in Emerald is transparent to the developers.

Some ideas of Emerald can also be found in Archipelago, such as the ability of shared objects to exhibit behaviour (active objects). However, Emerald represents a rather extreme approach by sharing all objects. Archipelago, on the other hand, distinguishes between shared objects and objects local to a participant by means of shared object spaces. Contrary to Emerald, it also allows the configuration and encapsulation of this boundary.

Distributed Shared Objects

The approach described by Homburg et al. uses distributed shared objects to hide network communication from parts of distributed applications [22], which is also a goal of the Archipelago platform. Shared objects in this approach consist of four parts: a control object, a semantics object, a replication object, and a communication object. The semantics object implements the actual functionality of the object, the replication and communication objects define the distribution and replication strategies of the object's data, and the control object acts as interface to local objects. For example, a shared object could be replicated and the effects of method executions synchronized; or, alternatively, it could be located centrally and methods could be executed as remote procedure calls. However, Homburg et al. only describe the general concepts of their approach. They leave many open questions and do not provide an implementation of their concepts.

Generally, the ideas presented by Homburg et al. have many things in common with Archipelago's: Developers can hide and encapsulate the

network communication and distribution boundary in the implementation of shared objects, but Homburg et al. do not discuss how this could be realized in detail. Similar to Archipelago, objects in their approach are implemented as local ones, that is, the semantics part of the shared object, and then later shared with other participants by choosing or generating the other parts. However, shared objects are passive and Homburg et al. do not discuss the relationships or interactions between multiple shared objects. They also do not use the concept of a shared object space to deploy the shared objects to, as Archipelago does, nor do they intend to use shared objects to implement coordination mechanisms.

GemStone: An Object Database

The GemStone system [10] provides a distributed Smalltalk runtime for the purpose of storing large sets of objects. It divides the system into server and client parts; the server manages and provides access to a shared object repository, which clients can access remotely via a connector library called GemBuilder. GemStone is closer to an object database management system than to a system for coordination between parts of a distributed application: GemStone's object repository is persistent and changes to shared objects are applied in transactions. GemStone provides an object-oriented language based on Smalltalk to define, modify, and query objects in the repository. For clients written in Smalltalk, this allows a mostly automatic mapping between database entities and entities in the client runtime. They can use the GemBuilder library to store client objects into the repository, import objects from the repository into the client runtime, or transparently replicate or proxy accesses to shared objects. Users can also customize the mapping between local client objects and repository objects.

Compared to Archipelago, the GemStone system provides many more guarantees about objects in the shared space and uses complex mechanisms to establish them: For example, shared objects can only be modified in transactional contexts. GemStone also explicitly separates the repository from external client applications, while Archipelago's shared object space is a part of the applications' runtime system. The use cases of the two platforms are different, too: GemStone is mainly used to store and query persistent data that has to be retained over multiple application runs, while Archipelago aims to support transient communication and coordination,

as well as storing short-lived data and executing shared application logic. We argue that GemStone's complexity is too large for such use cases and a simpler solution, such as Archipelago, is better suited.

5.3.2 Future Work

We have already identified a few topics of future work that concern potential solutions to the open questions of the Archipelago/S prototype: First, we are investigating whether static or dynamic code analysis or sandboxing can help prevent shared state inconsistencies that may result out of our boundary object design. Second, we would like to implement and evaluate a more customizable generic event mechanism that supports custom matching strategies and event handler definitions as an alternative to direct interaction with objects outside the shared space. Third, evaluating a passing-by-migration parameter passing policy and per-method or per-message policy configurations remain topics of future work, too.

Other possible features that could be investigated include replacing the centralized message router of TeaTime by a fully peer-to-peer solution and adding support for the automatic deployment of the classes (and other source code) of shared objects. Interesting ideas to realize the latter include using separate class definitions for objects in the shared space and local objects and sharing these class definitions between all participants, while replicating any changes to them.

Another important area of future work regarding Archipelago/S concerns its evaluation. We would like to evaluate our prototype in more complex use cases that involve multiple different shared spaces and coordination structures, such as sketched in the stock market example presented in Section 2.2.1. Furthermore, a more detailed performance and scalability evaluation of our prototype is necessary. In particular, we would like to measure the overhead of the replicated execution of messages compared to locally executed ones and the influence of the number of machines participating in a shared space on the performance. In order to better judge the usability of Archipelago/S, we are also planning on developing equivalent versions of ArchSpaces and SpaceFight using different approaches and technologies.

Regarding our platform design itself, topics of future work include evaluating different architectural approaches to implement the platform design, such as a centralized or distributed architecture, and their comparison

with Archipelago/S, addressing the issue that the design itself is limited to a single programming language (for example, through evaluation of code generation mechanisms), and extending the design with other features, such as persisting shared objects or garbage collection mechanisms.

5.4 Summary

In this chapter, we evaluated our Archipelago/S prototype through discussing the implementation of an exemplary coordination model and distributed application based on Archipelago/S: the ArchSpaces tuple space and SpaceFight game. In this context, we showed how Archipelago's concepts can be used to extend ArchSpaces with application-specific functionality and combine it with another data structure for use in SpaceFight. We also showed that moving parts of SpaceFight's application logic into the shared space can simplify its implementation.

In this process, we observed that shared objects can be developed like local ones and later easily modified for deployment into the shared space. Furthermore, Archipelago/S allowed us to configure and encapsulate the distribution boundary within the implementation of the shared objects, making the interaction with them simple. However, we also experienced that the limitations of our prototype, such as limited parameter passing policies or the dangers of extending boundary objects, can make working with Archipelago/S more difficult than necessary.

We discussed these and other issues in detail and, subsequently, compared Archipelago with related approaches and outlined topics of future work.

6 Conclusion

In this thesis, we presented and evaluated the Archipelago platform, our approach for implementing component interaction mechanisms in distributed applications, such as coordination models or shared data structures. We first discussed some of the coordination, communication, and data sharing technologies typically used in distributed applications. We described that technologies are often extended, customized, and combined to fit application-specific needs, but observed that custom extensions are often difficult to implement and combining multiple technology implementations can increase application complexity.

Archipelago allows the implementation of interaction mechanisms by means of object-oriented principles in the same programming language as the distributed application by building on shared object spaces between parts of the application. By employing the principles of object-orientation, application developers can then easily reuse, extend, and combine different technologies to fit application needs. Additionally, developers can give the responsibility for the execution of certain parts of the application functionality into the platform by implementing custom shared objects.

Our prototype implementation of the platform adapts Croquet's replication technology for the use in distributed applications consisting of computationally different components. With Croquet's Islands at its heart, Archipelago/S provides an efficient way to implement the platform design with replicated object spaces. Further, Archipelago/S gives developers a convenient way to encapsulate and configure the distribution boundary within shared objects by means of boundary objects and auto message makers.

We evaluated our platform design and prototype through implementing an exemplary coordination model, the ArchSpaces tuple space, and a distributed application, the SpaceFight game, which uses Archipelago's concepts to use and extend ArchSpaces, to combine it with a custom shared data structure, and to push functionality into the shared space. ArchSpaces and SpaceFight demonstrate that the Archipelago platform can

make the development of interaction mechanisms and simple distributed applications easy and convenient.

However, the discussion and evaluation of our platform and its prototype also surfaced several issues and open questions, which require further investigation. Primarily, we noted three problematic areas: First, the encapsulation of the distribution boundary is important for reusable shared objects, but our boundary object solution risks inconsistent shared object state by allowing local functionality to access shared state and functionality. Second, the direct approach to interact with local objects, which we employ for a custom event handling mechanism in ArchSpaces, may allow dependencies between shared and local objects that our platform design forbids. Third, the parameter passing policies should also support migrating objects and more flexible configurations.

Our next steps include evaluating solutions and alternative approaches to these issues, comparing Archipelago/S with distributed or centralized approaches to implement the platform design, and providing a more detailed and comparative performance evaluation for our platform prototype.

Bibliography

[1] Richard M. Adler. "Distributed Coordination Models for Client/ Server Computing". In: *Computer* 28.4 (1995), pp. 14–22.

[2] Gul Abdulnabi Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press Cambridge, MA, USA, 1986. 144 pp. ISBN: 0262010925.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN: 0201000237 978-0201000238.

[4] S. Ahuja, N. Carriero, and D. Gelernter. "Linda and Friends". In: *Computer* 19.8 (Aug. 1986), pp. 26–34. ISSN: 0018-9162. DOI: 10.1109/MC.1986.1663305.

[5] Saeed Alaei, Mohammad Toossi, and Mohammad Ghodsi. "Skip-Tree: A Scalable Range-Queryable Distributed Data Structure for Multidimensional Data". In: *Algorithms and Computation*. Ed. by Xiaotie Deng and Ding-Zhu Du. Lecture Notes in Computer Science 3827. Springer Berlin Heidelberg, Jan. 1, 2005, pp. 298–307. ISBN: 978-3-540-30935-2, 978-3-540-32426-3.

[6] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". In: *IEEE Transactions on Software Engineering* 18 (1992), pp. 190–205.

[7] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. "The LighTS Tuple Space Framework and Its Customization for Context-Aware Applications". In: *Web Intelligence and Agent Systems* 5.2 (Jan. 1, 2007), pp. 215–231.

[8] Guy Bieber. "Introduction to Service-Oriented Programming". In: *Openwings*. 2001.

[9] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. "Extensible Markup Language (XML)". In: *World Wide Web Consortium Recommendation REC-xml-19980210* (1998).

[10] Paul Butterworth, Allen Otis, and Jacob Stein. "The GemStone Object Database Management System". In: *Communications of the ACM* 34.10 (Oct. 1991), pp. 64–77. ISSN: 0001-0782. DOI: 10.1145/125223.125254.

[11] Nicholas Carriero and David Gelernter. "Linda in Context". In: *Commun. ACM* 32.4 (Apr. 1989), pp. 444–458. ISSN: 0001-0782. DOI: 10.1145/63334.63337.

[12] James O. Coplien, Douglas C. Schmidt, et al. *Pattern Languages of Program Design*. Vol. 58. Addison-Wesley Reading, 1995.

[13] Douglas Crockford. *The Application/Json Media Type for Javascript Object Notation (json)*. IETF RFC 4627, 2006.

[14] Stéphane Ducasse, Thomas Hofmann, and Oscar Nierstrasz. "OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces". In: *Coordination Languages and Models, LNCS 1906*. Springer, 2000, pp. 1–19.

[15] Anders Fongen. *A SmallSpaces Programming Tutorial*. The Norwegian School of Information Technology, 2006.

[16] The Apache Software Foundation. *Apache ActiveMQ*. URL: http://activemq.apache.org/ (visited on 2014-08-10).

[17] The Apache Software Foundation. *Apache Qpid*. URL: https://qpid.apache.org/ (visited on 2014-08-10).

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[19] David Gelernter. "Generative Communication in Linda". In: *ACM Transactions on Programming Languages and Systems* 7.1 (Jan. 1985), pp. 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433.

[20] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In: *OSDI*. Vol. 12. 2012, p. 2.

[21] Object Management Group. *The CORBA Standard*. URL: http://www.omg.org/spec/CORBA/Current/ (visited on 2014-08-10).

[22] Philip Homburg, Maarten Van Steen, and A. Tanenbaum. "Distributed Shared Objects as a Communication Paradigm". In: *Proceedings of the 2nd Annual ASCI Conference*. 1996, pp. 132–137.

[23] Google Inc. *Protocol Buffers: Google's Data Interchange Format. Documentation and Open Source Release*. URL: https://code.google.com/p/protobuf/ (visited on 2014-08-10).

[24] Oracle Inc. *Java Remote Method Invocation - Distributed Computing for Java*. URL: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html (visited on 2014-08-10).

[25] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *In Proceedings OOPSLA '97, ACM SIGPLAN Notices*. ACM Press, 1997, pp. 318–326.

[26] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-Grained Mobility in the Emerald System". In: *ACM Transactions on Computer Systems* 6.1 (Feb. 1988), pp. 109–133. ISSN: 0734-2071. DOI: 10.1145/35037.42182.

[27] M. Frans Kaashoek and David R. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table". In: *Peer-to-Peer Systems II*. Ed. by M. Frans Kaashoek and Ion Stoica. Lecture Notes in Computer Science 2735. Springer Berlin Heidelberg, Jan. 1, 2003, pp. 98–107. ISBN: 978-3-540-40724-9, 978-3-540-45172-3.

[28] Deepali Khushraj, Ora Lassila, and Tim Finin. "sTuples: Semantic Tuple Spaces". In: *In Proceedings of the 1st International Conference on Mobile and Ubiquitous Systems*. 2004, pp. 267–277.

[29] Athanasios Konstantinidis, Antonio Carzaniga, and Alexander L. Wolf. "A Content-Based Publish/Subscribe Matching Algorithm for 2d Spatial Objects". In: *Middleware 2011*. Ed. by Fabio Kon and Anne-Marie Kermarrec. Lecture Notes in Computer Science 7049. Springer Berlin Heidelberg, Jan. 1, 2011, pp. 208–227. ISBN: 978-3-642-25820-6, 978-3-642-25821-3.

[30] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed Graphlab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the VLDB Endowment* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354.

[31] GigaSpaces Technologies Ltd. *Gigaspaces XAP*. URL: http://docs.gigaspaces.com/product_overview/ (visited on 2014-04-29).

[32] Ronaldo Menezes and Alan Wood. "Garbage Collection in Open Distributed Tuple Space Systems". In: *In Proc. 15th Brazilian Computer Networks Symposium — SBRC'97*. 1997, pp. 525–543.

[33] *Message Passing Interface Standard*. URL: http://www.mcs.anl.gov/research/projects/mpi/standard.html (visited on 2014-08-10).

[34] Bertrand Meyer. *Object-Oriented Software Construction*. Vol. 2. Prentice Hall New York, 1988. ISBN: 0136291554.

[35] Amy L. Murphy and Gian Pietro Picco. "Using Coordination Middleware for Location-Aware Computing: A Lime Case Study". In: *in Proc. of the 6 th Int. Conf. on Coordination Models and Languages (COORD04), LNCS 2949*. Springer, 2004, pp. 263–278.

[36] Hamid Nazerzadeh and Mohammad Ghodsi. "RAQ: A Range-Queriable Distributed Data Structure". In: *SOFSEM 2005: Theory and Practice of Computer Science*. Springer, 2005, pp. 269–277.

[37] George A. Papadopoulos and Farhad Arbab. "Coordination Models and Languages". In: *Advances in Computers* 46 (1998), pp. 329–400.

[38] Julien Pauty, Paul Couderc, Michel Banatre, and Yolande Berbers. "Geo-Linda: A Geometry Aware Distributed Tuple Space". In: *Proceedings of the 21st International Conference on Advanced Networking and Applications*. AINA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 370–377. ISBN: 0-7695-2846-5. DOI: 10.1109/AINA.2007.74.

[39] Gian Pietro Picco, Davide Balzarotti, and Paolo Costa. "LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications". In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. SAC '05. New York, NY, USA: ACM, 2005, pp. 413–419. ISBN: 1-58113-964-0. DOI: 10.1145/1066677.1066775.

[40] *RabbitMQ - Messaging That Just Works*. URL: https://www.rabbitmq.com/ (visited on 2014-08-10).

[41] David Patrick Reed. "Naming and Synchronization in a Decentralized Computer System." Thesis. 1979. Ph.D.–Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science. PhD thesis. Massachusetts Institute of Technology, 1978.

[42] Robert Sedgewick and Kevin Daniel Wayne. *Algorithms*. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN: 0-321-57351-X 978-0-321-57351-3.

[43] W. Segall and D. Arnold. "Elvin Has Left the Building: A Publish/-Subscribe Notification Service with Quenching". In: *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, September 1997*. http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html. 1997.

[44] D.A Smith, A Raab, D.P. Reed, and A Kay. "Croquet: A Menagerie of New User Interfaces". In: *Second International Conference on Creating, Connecting and Collaborating through Computing, 2004. Proceedings*. Jan. 2004, pp. 4–11. DOI: 10.1109/C5.2004.1314362.

[45] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. "Croquet - a Collaboration System Architecture". In: *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.* IEEE, 2003, pp. 2–9.

[46] David A. Smith, Andreas Raab, David P. Reed, and Alan Kay. *Croquet Programming - a Concise Guide*. 2006.

[47] A. Vijay Srinivas and D. Janakiram. "Scaling a Shared Object Space to the Internet: Case Study of Virat." In: *Journal of Object Technology* 5.7 (2006), pp. 75–95.

[48] A Vijay Srinivas, D Janakiram, and Raghevendra Koti. *Virat: An Internet Scale Distributed Shared Object, Event and Service Space*. Technical Report IITM-CSE-DOS-2004-03, Distributed & Object Systems Lab, Indian Institute of Technology, Madras, 2004.

[49] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Pearson Prentice Hall, 2007. ISBN: 0132392275 9780132392273.

[50] *The Open Group Base Specifications Issue 7, 2013*. URL: http://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 2014-08-10).

[51] "To Allow One or More Processing Systems to Process Client Requests". US5797005 A. James W. Bahls, George S. Denny, Richard G. Hannan, Janna L. Mansker, Bruce E. Naylor, Karen D. Paffendorf, Betty J. Patterson, Sandra L. Stoob, Judy Y. Tse, and Anu V. Vakkalagadda. Aug. 18, 1998.

[52] Jim Waldo. *Javaspaces Specification 1.0*. Sun Microsystems, 1998.

[53] Jim Waldo. "The Jini Architecture for Network-Centric Computing". In: *Communications of the ACM* 42.7 (July 1999), pp. 76–82. ISSN: 0001-0782. DOI: 10.1145/306549.306582.

[54] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. IEEE Micro, 1994.

[55] S. Weinbrenner, A Giemza, and H.U. Hoppe. "Engineering Heterogeneous Distributed Learning Environments Using Tuple Spaces as an Architectural Platform". In: *Seventh IEEE International Conference on Advanced Learning Technologies, 2007. ICALT 2007*. July 2007, pp. 434–436. DOI: 10.1109/ICALT.2007.139.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den

Eric Seckler