Archipelago – A Research Platform for Component Interaction in Distributed Applications

Conference Paper · October 2014

DOI: 10.13140/2.1.2425.8240

2 authors, including:

Eric Seckler
Hasso Plattner Institute
4 PUBLICATIONS **2** CITATIONS

SEE PROFILE

Archipelago

A Research Platform for Component Interaction in Distributed Applications

Eric Seckler
eric.seckler@hpi.uni-potsdam.de

Robert Hirschfeld
robert.hirschfeld@hpi.uni-potsdam.de

Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany

ABSTRACT

Distributed applications consist of different parts, which interact across distribution boundaries to achieve a common goal. The complexity of these interactions can vary within a single application and different interaction tasks require different mechanisms to communicate, coordinate, or share data or computation. We propose and evaluate Archipelago, our research platform to investigate and better understand object-oriented interaction in distributed applications. Archipelago is based on a shared object space that is replicated between application parts, a replication technology adopted from the Croquet project. We evaluate Archipelago by implementing illustrative examples and argue that our approach allows application parts to conveniently share structured data and computation and enables the implementation of reusable and extensible interaction mechanisms.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Netw.**]: Distributed Systems—*Distributed Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.2.11 [**Software Eng.**]: Software Architectures

Keywords

distributed applications, replication, object space, coordination, distribution boundary, tuple space, Croquet, Squeak

1. INTRODUCTION

A distributed application, such as a multi-player networked game, typically consists of multiple parts, which work together to achieve a common goal. For this purpose, they need to coordinate their computation and share or exchange data with each other. Interaction complexity may vary within a single application. For example, sharing positions of entities in a game can be easily solved with a general-purpose mechanism, such as a tuple space [3], but more complex shared data, such as sorted, range-queryable game statistics, may require a custom solution.

REBLS '14 Portland, USA

In this paper, we evaluate an object-oriented approach to implementing these interactions, which is based on a replicated object space within the runtime system of the different application parts. The space can contain multiple active objects that can interact with each other and local objects. These shared objects can be used to implement general-purpose or application-specific shared data structures or coordination mechanisms and other shared functionality.

We present our research platform *Archipelago* that is based on Croquet [11], which keeps object spaces called Islands [12] consistent by replicating object computation instead of object state, that is, the operations performed on shared objects are synchronized and executed in the same order on all object replicas. We repurpose and adapt Croquet's Islands for general use in distributed applications comprised of computationally different parts.

We argue that our approach to object-oriented interactions in distributed applications allows different application parts to share structured data and computation and enables the implementation of reusable, extensible, and customizable coordination mechanisms and shared data structures.

Our contributions are as follows:

- Our approach is based on the transfer of Croquet's Islands for use in distributed applications with computationally different parts. In this context, we investigate the use of replicated active objects for the interaction between such parts. Contrary to purely passive replicated data, this allows us to exploit the benefits of object-orientation, for example, to customize replicated object behaviour.
- The implementation of Archipelago uses Croquet's Islands at its heart and adapts and extends them for our use case. We particularly focus on how to provide a convenient way for interaction between shared and local objects. 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.
- We evaluate our approach and the Archipelago platform by implementing a peer-to-peer multi-player networked game as an example application. We also show how Archipelago can be used to implement a reusable and extensible tuple space and how it can be used for coordination and data sharing in the game.

The remainder of this paper is structured as follows: Section 2 introduces SpaceFight, an example application that we use to motivate our approach. Section 3 provides context by discussing different implementation architectures for

a shared object space and the Croquet project. Section 4 describes the ideas of our approach and Section 5 its implementation. Section 6 shows how Archipelago can be used to implement SpaceFight, which utilizes ArchSpaces, a reusable, extensible tuple space for Archipelago. It further evaluates and discusses our approach and its limitations. Section 7 presents related approaches. Finally, Section 8 concludes this paper.

2. MOTIVATING EXAMPLE

We introduce SpaceFight, a networked multi-player game, to discuss the ways in which different parts of a distributed application communicate, coordinate and share data. SpaceFight borrows from games such as SpaceWar or Asteroids. Each player navigates a ship through an asteroid-filled region in space. A collision with an asteroids 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.

We group the interaction mechanisms between the different components into shared state, shared computation, and general coordination tasks. 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. 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; because of this, we regard them as *shared*. The components not only need to coordinate the execution of shared computations, but also have to solve other coordination tasks, such as recognising and reacting to game state changes, changes in movement vectors of entities, or entity destructions.

Some of these interaction activities are rather simple, while others are more complex. For example, the game state can be held in a simple key-value or relational data structure and the associated coordination tasks can be solved by detecting changes to this data structure. A tuple space [5] is a simple data and coordination structure that can be used for these purposes: The game state can be stored as tuples and dependents can be notified about changes to these tuples through event-based subscriptions. However, not all activities are this simple. The participants of our game 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, a custom tree-based data structure that supports such operations and duplicate detection is more useful to store the statistics than a tuple space.

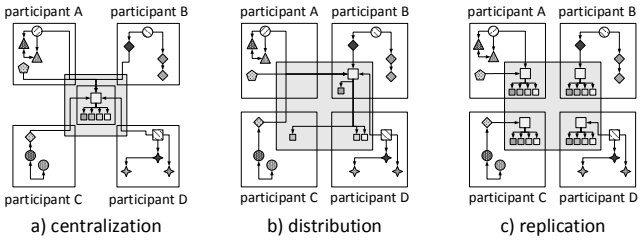

Figure 1: Implementation architectures for shared objects in distributed applications. In each case, shared data is accessed by different parts of the application. Different objects are represented by different shapes or colors, relationships between them by arrows. The gray area in the center of each picture illustrates the shared part of the application, which can be accessed by crossing the distribution boundary, for example, via remote procedure calls.

In summary, the participants share simple data and coordination tasks, which can be solved by a general-purpose model such as a tuple space, and more complex ones, for which custom data structures are better suited. Furthermore, some computations are shared by all participants of the game. We argue that a shared object space, which can contain active objects, allows us to conveniently implement custom shared structures and computations. We can also use this object space to implement general-purpose structures, such as a tuple space, while enabling their extension and customization to application-specific needs.

3. BACKGROUND

Before introducing our proposal, we provide some context by describing some of the architectural choices we considered for implementing shared objects and Croquet's Islands [11], which form the base for our implementation.

3.1 Architectures for Sharing Objects

For the design of a distributed application that uses shared objects to communicate, coordinate, and share data, three different architectures for the distribution of shared objects come to mind, which can also be combined: centralization, distribution, and replication (Figure 1).

Centralization. In a centralized architecture (Figure 1a), all shared objects reside on a dedicated server. Operations on these objects are usually executed as remote procedure calls or their respective language equivalent. Therefore, the other participants access the objects through some kind of proxy object in their local runtime system, which performs the remote calls and the underlying network message exchange with the server. As such, this approach is relatively easy to implement, however the central server is both a bottleneck for scalability and a single point of failure.

Distribution. The dedicated server of the centralized architecture can be replaced by distributing the objects amongst the participants, so that each shared object resides with a particular participant. In this distributed architecture (Figure 1b), participants can interact with shared objects that reside locally via normal message sends of the programming language and use proxies and remote procedure calls to access objects that are located remotely. While this approach is much more scalable than the centralized approach in terms of performance, it is even less reliable, as each participants becomes a single point of failure for the objects it hosts.

Replication. An alternative approach is to replicate all shared objects with all participants (Figure 1c). In this case, participants can access objects locally, but changes to objects have to be synchronized to all other participants in order to maintain consistency in the system. Therefore, participants typically also access shared objects through a proxy object, which ensures synchronization and, with that, consistency when performing operations on a shared object. This replicated architecture is limited in scalability, as the consistency synchronization involves all participants, however, it is very reliable and failure-tolerant, because copies of all objects reside with each participant.

3.2 The Croquet Collaboration System

Croquet [11] aims to provide a 3D real-time collaboration platform for multiple participating users. 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 is based on the Squeak/Smalltalk [6] environment and uses a replication mechanism called *TeaTime* [11], 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 [10] and intended to be a pure peer-to-peer architecture, however, the current implementation of TeaTime in Croquet requires a central message router to perform the synchronization tasks.

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.

4. ARCHIPELAGO

We argue that a replicated shared object space constitutes a good abstraction to implement extensible coordination mechanisms as well as shared data structures and shared computations in a distributed application, such as the SpaceFight game presented in Section 2. Shared active objects can combine both shared data and shared computations. By confining these shared objects to a separate region in the application's runtime system, that is, a shared object space, the distribution boundary can be conveniently encapsulated by the platform's infrastructure. Furthermore, replicating shared objects between the participants of the shared space makes these objects independent of the individual participants' availability and, therefore, more reliable.

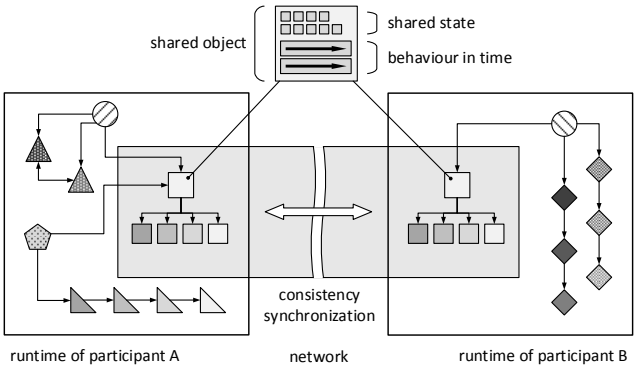

Figure 2: Shared object space within the runtime systems of two participants: The objects within the shared space are replicated and consistent for both participants. All objects outside this space are local to the respective participant.

To investigate this proposal, we present *Archipelago*, our research platform for communication, coordination, and sharing data and computation in distributed applications. Shared objects are deployed into a replicated object space and can then be accessed by all participants. With Croquet's Island as its core, Archipelago's participants always share a consistent view on objects in this space and can interact with shared objects as if they were local. These objects are active objects, that is, they can exhibit behaviour, and are independent of individual participants. Figure 2 illustrates such a shared space in the Archipelago platform. Many ideas behind Archipelago are taken from or influenced by Croquet's Islands and TeaTime technologies. As we discuss in Section 5, Archipelago adapts these ideas for use in distributed applications comprised of computationally different parts.

4.1 Interacting with Shared Objects

Objects local to one particular participant have to be able to interact with shared objects by sending messages to them. The effects of these message sends have to be replicated across all copies of a shared object, that is, the messages have to cross a *distribution boundary*. Regarding this, two aspects are of particular interest: the way in which the distribution boundary is crossed and the way arguments and return values are passed for such message sends.

Distribution Boundary. 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 [14]. We argue that while it is true that crossing the distribution boundary should be explicit, the main responsibility for this should lie with the developer of a shared object. Therefore, the Archipelago platform allows to encapsulate 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, for example, to handle failures. This makes it possible to implement reusable ob-

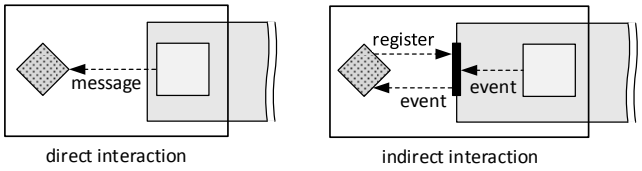

Figure 3: Direct vs. indirect interaction with local objects. We show the messages exchanged.

ject structures, such as coordination mechanisms and data structures, that encapsulate the distribution boundary from their users, while making it explicit to their developers.

Parameter Passing. We distinguish three different ways, in which arguments to and return values from message sends can be passed over the distribution boundary: by reference, by value, or by migration. If passed by reference, the parameter will become a reference to an object valid on any participant in the network. In our platform, this is only possible with shared objects, as references to local objects are only valid on a single participant. Alternatively, a shared copy of the argument value could be created and this copy passed instead. In this case, the parameter value object will be a separate object from the copy that is passed, that is, later changes to this (local) object will not be reflected in its copy and not visible to other participants and vice versa. Another possibility is to pass the argument by copying it into the shared space and replacing the original value object with this migrated copy. This way, later changes performed on the object are reflected in the shared space.

The Archipelago platform implements only passing by reference for arguments that are shared objects and passing by value for arguments that are objects local to one participant. Return values from replicated methods are passed by value or reference: Objects of basic or immutable types are passed by value, objects of other types by reference. However, we have noticed that the intended parameter passing behaviour can also vary depending on the message sent. Therefore, we are still investigating whether a different parameter passing policy is better suited, which integrates object migration and allows the developer to specify the way that arguments for calls of a specific method are to be passed.

4.2 Interacting with Local Objects

Not only local objects need to interact with shared ones, but also shared objects have to be able to send messages to local ones. Coordination mechanisms that allow for event-based notifications of subscribers are an example for this: They have to send messages as callbacks to local subscribers. We are considering two different concepts for this interaction: either based on direct or on indirect communication, as illustrated in Figure 3.

For the direct approach, shared objects need to know of local objects and can send them messages directly. This can be implemented by allowing shared objects to maintain references to local objects from their local side. Functionality on the replicated side of a shared object can then have side effects affecting these references on the local sides of all its replicated copies. This direct approach allows flexible interaction with local objects, but means that the local side of shared objects has to be exposed to the replicated side.

The indirect approach, on the other hand, 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 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, the Archipelago platform allows shared objects to use both direct and indirect communication for interaction with local objects.

5. IMPLEMENTATION

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 implementation is based on replicating computation instead of data and, for this purpose, makes use of the TeaTime replication technology from the Croquet system. 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 [12].

For use in Archipelago, we separated the Island and TeaTime technology from the Croquet-specific use-case of fully replicated 3D collaboration applications 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 for 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.

We first sketch the implementation of Islands and TeaTime and describe how shared objects are interacted with in Croquet and then discuss our modifications and extensions that adapt these mechanisms to our use case.

5.1 Replicated Objects via TeaTime

As described in Section 3.2, objects in Croquet are contained within an object space called Island and their copies are held consistent by replicating computation, that is, executing the operations on these objects on all copies. Therefore, the order of operations has to be synchronized and all operations have to be deterministic.

The TeaTime architecture implements this mechanism. 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 that are scheduled to be executed at some point in the future. It also

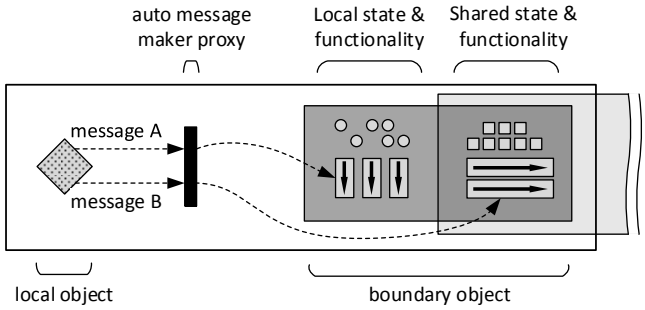

Figure 4: Boundary objects and auto message makers. Message A is executed on the local side of the shared object, message B is replicated.

implements a mechanism that allows participants to join an existing Island at any point in its execution. More details about this can be found elsewhere [12].

5.2 Shared Objects in Croquet

Because message sends to shared objects have to be intercepted and replicated, shared objects are accessed from outside the Island via a proxy object called *FarRef*. FarRef proxies allow users of a shared object to send messages to a shared object that are either executed locally or replicated: By sending a FarRef the message `future` followed by the message to the shared object, the message is executed replicated. A similar mechanism allows to execute a method locally, which can be used, for example, to access local object values without replicating the respective message to all copies of the object.

Replicated message sends always return a new FarRef object, which is a placeholder for the return value of the message. Once the message is executed by the Island, the FarRef will be filled with this value. Users can send more replicated messages to this FarRef without waiting for the message to be executed, but if they need to access a return value that is of basic type, they need to wait for the execution to take place by sending the FarRef the message `wait` and then retrieve the actual value by sending the `value` message.

For illustration, consider this interaction with a shared Point object in Croquet to set and retrieve a coordinate:

```
sharedPoint future x: 10.
farRef := sharedPoint future x.
value := farRef wait; value.
```

5.3 Encapsulating the Distribution Boundary

Interacting with shared objects through FarRefs 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 is of basic type and requires sending the `wait` and `value` messages to access it. In order to conveniently use shared objects, Archipelago adds support for encapsulating the distribution boundary within the implementation of the shared object itself. Therefore, we propose two new concepts: boundary objects and auto message makers, illustrated in Figure 4.

Boundary Objects. Boundary objects are shared objects that encapsulate the distribution boundary. A boundary object has two sides to it: a local and a shared one. Both sides can contain both state and functionality. 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. As local functionality is not executed replicated, it must not modify any shared state. Currently, we leave the responsibility for ensuring this to the developer; for a discussion of this issue see Section 6.4. 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 1.

Auto Message Makers. Auto message makers form the counterpart to boundary objects and function as additional proxy objects around FarRefs. 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. 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 basic type. In consequence, users of shared objects that are accessed through a auto message maker no longer have to use the special interface of FarRefs with `future`, `wait` and `value` messages, and the interaction with the shared Point could look just like the interaction with a local one:

```
sharedPoint x: 10.
value := sharedPoint x.
```

5.4 Interacting with Local Objects

As described in Section 4.2, Archipelago allows two different ways for shared objects to interact with local objects: by direct and indirect communication. Croquet's Island implementation already supports an indirect way for this interaction: 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.

However, to allow more flexible interaction with local objects, Archipelago also adds a way for shared objects, particularly boundary objects, to interact directly with local objects. Therefore, references to local objects can be passed to shared objects via messages that are executed locally and can be stored in the local state of the boundary object. The shared functionality of this object can then access these references and exhibit side effects on these. This way, shared objects can, for example, implement a custom publish-subscribe mechanism, as we will show with our tuple space implementation in the following section.

6. EVALUATION

We evaluate the Archipelago platform by using it to implement the SpaceFight game described in Section 2. The game uses ArchSpaces, a tuple space implementation for our platform, for basic data sharing and coordination tasks and further deploys custom structures and functionality to the shared object space for more complex tasks.

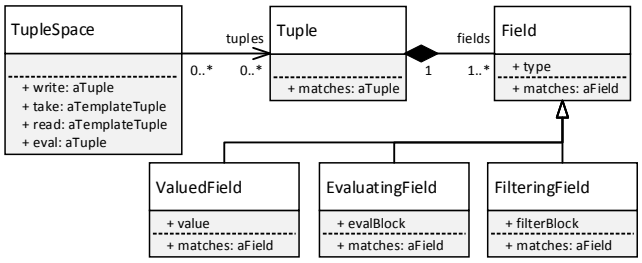

Figure 5: Class model of ArchSpaces.

6.1 ArchSpaces: Tuple Spaces in Archipelago

We built ArchSpaces as a general-purpose, reusable tuple space for Squeak, which we then deployed to Archipelago.

A Simple Tuple Space for Squeak. ArchSpaces implements the `write`, `take`, `read` and `eval` operations of the Linda interface [5]: `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 and further extend the interface with features for event-based interaction: Users can register event listeners for newly added or removed tuples matching a given template.

ArchSpaces is based on an object-oriented tuple space model, which is depicted in Figure 5. 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, a `ValuedField` filters for a specific value and a `FilteringField` allows filtering based on a custom filter represented as a code block.

The general-purpose implementation of ArchSpaces uses a simple ordered list to store all tuples contained in the space, and performs a linear scan when asked to retrieve a tuple.

Deploying ArchSpaces to Archipelago. 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 the `TupleSpaces` 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 local objects.

For the implementation of the tuple space boundary object, we specified which operations on a tuple space should be performed locally: The `read` operation can be performed without replicating its execution, as it does not have any side-effects on the state of the tuple space. Further, the event listener functionality 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 (Listing 1). All other operations on the tuple space are executed 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.

Listing 1: Example for labelling of TupleSpace operations: read is executed locally, write replicated.

```
TupleSpace>>write: aValueTuple
        self tuples add: aValueTuple.
        self fireWriteListenersFor: aValueTuple

TupleSpace>>read: aTemplateTuple
        <executeLocally>
        ↑ self tuples find: aTemplateTuple
```

However, there are two limitations of this deployment: 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, but functionality inside the Island has to be deterministic and cannot use unordered asynchrony. Furthermore, code blocks cannot easily be serialized: 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.

To overcome part of these limitations, we further extended ArchSpaces with support for 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. Services can also replace evaluating tuples, because, as shared objects, they can exhibit behaviour similar to a process evaluating a tuple. However, our current implementation of services is limited in that they cannot use blocking operations: As the services are executed within the Island's message processing loop, blocking operations currently result in blocking further messages to be processed. We plan to investigate alternative approaches to implement the blocking operations for services, so that this restriction can be lifted.

6.2 SpaceFight: An Example Application

We use Archipelago and ArchSpaces to implement the SpaceFight game described in Section 2. For the purpose of this game, we customized the ArchSpaces implementation and deployed application-specific functionality to Archipelago. We first sketch the architecture of the game implementation and then discuss the customization of ArchSpaces and the game's shared functionality.

Implementation Architecture. The game architecture is illustrated in Figure 6. 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 discussed in Section 2, 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

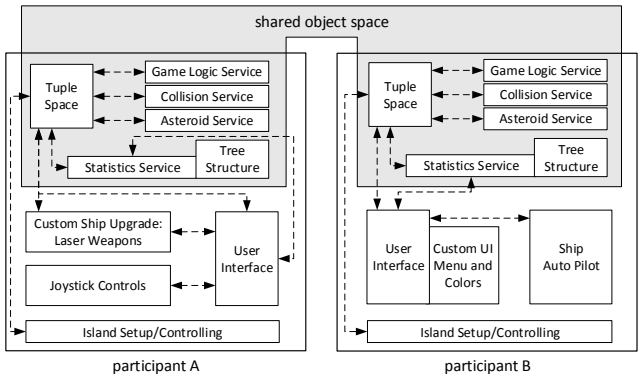

Figure 6: SpaceFight game architecture. Here, we show the shared and local components in a setup with two participants and their interactions.

describing entity collisions, which are written to the space by the collision detection service.

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 router of the Island, however, this player then becomes a single point of failure, therefore, it is also possible to dedicate a different participant for this role. Here, the central router of the croquet architecture becomes a major limitation of the architecture; we discuss this further in Section 6.5.

An Application-Specific Tuple Space. 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. We implemented these extensions by subclassing the `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 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. Tuple spaces are therefore often accessed through customized proxy objects that allow developers to specify application-specific operations on the tuple space. 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 within a single replicated operation, which can reduce the underlying network communication.

Application-Specific Functionality in the Island. As described in Section 2, our game contains functionality that all participants share the responsibility for and whose effects have to be consistent amongst all participants. Archipelago 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, randomly spawn asteroids on the map and perform the task of detecting collisions.

We further use a custom shared tree structure for storing SpaceFight statistics, as explained in Section 2. 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.

6.3 Discussion

With our implementation of the Archipelago platform that is based on Croquet's replication mechanism, sharing objects is simple and efficient: Objects are deployed into an Island, replicated at all participants and held consistent by replicating computations on them. Generally, any object can be shared and labelling or tagging for the deployment inside an Island is only necessary for advanced boundary objects with local and shared sides. Shared objects are independent of the availability of individual participants, which is important for applications like our SpaceFight game, where participants can freely join or leave the Island.

By sharing objects, Archipelago allows to share structured data between different parts of a distributed applications. As objects in Archipelago can further be active objects and exhibit behaviour, Archipelago makes it also possible to deploy shared functionality to the Island. This is especially convenient for peer-to-peer application architectures, because no additional coordination between the peers is necessary for determining who executes this functionality.

With the help of boundary objects and auto message makers, developers of shared objects can encapsulate the distribution boundary in the interaction with shared objects within their implementation. This is useful for the implementation of reusable coordination mechanisms or data structures in Archipelago, such as our ArchSpaces tuple space. Because these structures are implemented within the same runtime system and with object-oriented principles, they are also easily extensible to application-specific use cases, as we have shown by extending ArchSpaces for use in SpaceFight.

In SpaceFight, we use tuple spaces as main coordination medium, decoupling the different participants. We have found that tuple spaces provide a good abstraction for typical coordination tasks and are especially convenient when used with an event-based mechanism. By extending the tuple space with an application-specific interface, we could further reduce the data coupling of the participants and encapsulate tuple format knowledge.

6.4 Open Questions

During the implementation of Archipelago, we made a few choices that we feel need further evaluation and investigation. 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 [4].

A similar issue exists for the direct interaction of shared objects with local objects discussed in Section 4.2. Direct interaction gives shared objects more flexibility, 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 stays the same at all object replicas, even if these return values are different at different participants.

Another open issue is that of the way that parameters are passed into the shared space, discussed in Section 4.1. Our current implementation based only on passing local objects by value may not suffice. For example, in some cases, such as the dynamic addition of services at our tuple space implementation, passing by migration is 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. We also plan to evaluate different settings for the default behaviour.

6.5 Limitations

The most important limitation of the Archipelago platform is its restriction to a single programming language. While our focus has not yet been on portability, we believe it to be difficult to accomplish, as all participants need the source code of the shared objects to execute replicated methods. It may be possible to support multiple languages, for example, by automatic translations of the source code.

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.

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[10] can replace the central message router.

We also realize that we have yet to evaluate Archipelago regarding the topics of scalability, fault-tolerance, and performance, which are particularly important in the scope of distributed applications and constitute topics of future work.

7. RELATED WORK

A popular approach to object-oriented interaction in distribution systems, exercised, for example, by CORBA [8], Java RMI [9] and Jini [13], is based on a distributed architecture and is commonly referred to as the *distributed objects* approach. In these platforms, each participant can export objects by marking them as a special kind of object that can be accessed remotely and then share exported objects with other participants, usually by registering them with a so called service registry component [2]. This component maintains a list of objects with each of them providing some form of service and their service description, so that other participants can look up and find objects with the help of this description. That is, they can obtain remote references to shared objects through the service registry. 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 described before. 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. Archipelago's boundary objects have a similar purpose as RMI/Jini smart proxies, but do not separate the proxy from the shared object. The parameter passing semantics of RMI and Jini are also similar to Archipelago's - Shared objects are usually passed by reference and other objects by value. While in CORBA, the interface of shared objects has to be specified in a special interface definition language, and in RMI and Jini, it is specified as a separate Java interface, shared objects in Archipelago do not require any separate interface description and local state or functionality only has to be labelled for advanced shared objects. This allows users of Archipelago to easily share simple objects without further labelling or interface description.

Other approaches similar to Archipelago include languages for shared data structures, such as Orca [1], and mobile distributed objects, such as Emerald [7]. These languages typically have in common that the distribution boundary is completely transparent. The Orca language, for example, lets the application developer specify abstract object data types and their operations and interface with these objects

as if they were local, while its run time system transparently assures their distribution, synchronization and consistency between the participants. Contrary to Archipelago, shared objects in Orca are passive and their operations can only access the object's internal data.

In the Emerald system, 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. As in Orca, the distribution boundary in Emerald is transparent to the developers, too. Archipelago, on the other hand, makes this boundary explicit and allows its encapsulation. We also distinguish between shared objects and objects local to a participant.

8. CONCLUSION AND FUTURE WORK

In this paper, we present our approach to interaction and coordination between different parts of a distributed application based on a replicated object space shared between them. We evaluate *Archipelago*, our research platform implementing this approach, in the context of the *SpaceFight* game as an illustrative application. We also demonstrate how Archipelago can be used to implement custom shared data structures and computations and extend and customize *ArchSpaces*, a general-purpose tuple space, to the application-specific needs of SpaceFight.

During the project, we were reminded that encapsulating the distribution boundary from client or local objects is important when providing for shared or replicated objects. Furthermore, we try not to make the boundary completely transparent but embrace that the developer of shared objects should be able to customize access to them. However, as discussed in the evaluation, our boundary object implementation that supports this encapsulation currently does not prevent developers from breaking replicated objects' consistency. Furthermore, we have noted that parameter passing policies for the interaction with shared objects needs more flexibility and reasonable defaults.

To analyse these issues further, we are planning to evaluate Archipelago for building applications more complex than SpaceFight. For example, the distributed parts of such an application should be more versatile and require multiple scopes of interaction between different sets of participants. In this context, we would also like to evaluate Archipelago's capabilities regarding scalability and fault-tolerance, as well as performance aspects. Another idea that requires further evaluation is the use of replicated object spaces to implement the coordination of fully distributed data structures, such as a distributed hash table. We would like to find out whether replicated objects are suitable for this or which other interesting alternatives to investigate.

Acknowledgments

We would like to thank the late Andreas Raab for many engaging and insightful discussions throughout the years.

9. REFERENCES

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[2] G. Bieber, L. Architect, and I. Ci. Introduction to Service-Oriented Programming. In *Openwings*, 2001.

[3] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

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

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings OOPSLA '97*, pages 318–326, 1997.

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

[8] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 3.3)*, November 2012.

[9] Oracle Inc. *Java Remote Method Invocation - Distributed Computing for Java*.

[10] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. Thesis, Massachusetts Institute of Technology, September 1978.

[11] D. A. Smith, A. C. Kay, A. Raab, and D. P. Reed. Croquet—A Collaboration System Architecture. In *Conference on Creating, Connecting and Collaborating through Computing (C5)*, pages 2–9, 2003.

[12] D. A. Smith, A. Raab, D. P. Reed, and A. C. Kay. *Croquet Programming—A Concise Guide (Draft 0.14)*. Qwaq and Viewpoints Research Institute, 2006.

[13] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[14] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, 1994.