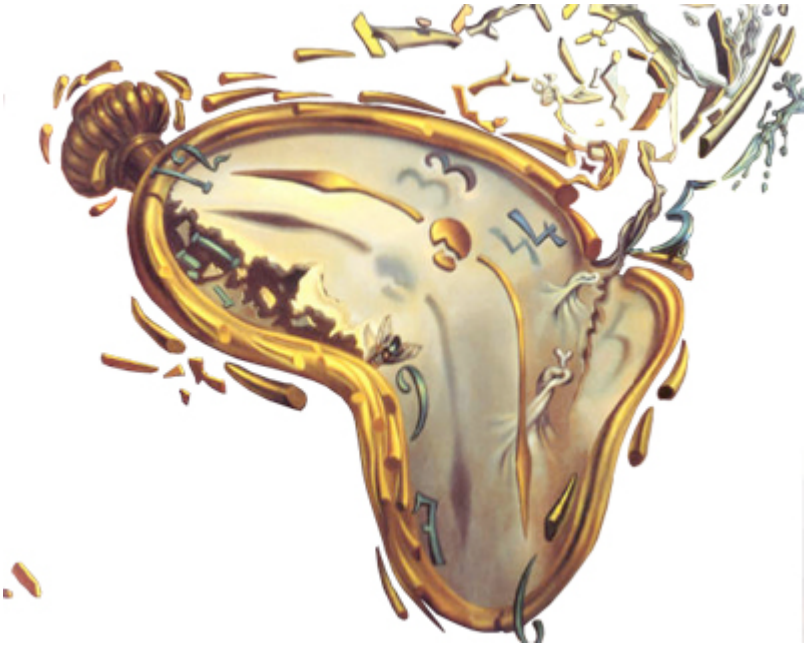
System Overview

From Croquet Consortium

Contents



Introduction

Croquet is a powerful new open source software development environment for creating and deploying deeply collaborative multi-user online applications on multiple operating systems and devices. Derived from Squeak, it features a peer-based network architecture that supports communication, collaboration, resource sharing, and synchronous computation between multiple users on multiple devices. Using Croquet, software developers can create and link powerful and highly collaborative cross-platform multi-user 2D and 3D applications and simulations - making possible the distributed deployment of very large scale, richly featured and interlinked virtual environments.

Every part of the system is designed around enabling real-time, identical interactions between groups of users. The architecture of Croquet actually makes it quite easy to develop collaborative applications without having to spend a lot of effort and expertise in understanding how replicated applications work. There are a number

of simple patterns and rules to remember, but otherwise, it is quite simple to quickly develop very powerful systems. The following sections are intended to provide a general overview of how the Croquet system actually works.

TeaTime and Islands are the basis for Croquet's replicated computation and synchronization. They are designed to support multi-user applications that can be scaled to massive numbers of concurrently interacting users in a shared virtual space. The TeaTime approach is designed to support multi-user applications that can be scaled to massive numbers of concurrently interacting users in a shared virtual space. The most directly visible part of this architecture is the TObject class which is used to define and construct subclassed Tea objects. A Tea object acts with the property that messages sent to it are redirected to replicated copies of itself on other users participating machines in the peer-to-peer network. All of the interesting objects inside of Croquet are constructed out of subclasses of TObject. This messaging protocol supports a coordinated distributed two-phase commit that is used to control the progression of computations at participating user sites. In this way, messages may be dynamically redirected to large numbers of users while maintaining the appropriate deadline based scheduling. Thus, TeaTime is designed to allow for a great deal of adaptability and resilience, and works on a heterogeneous set of resources. It is a framework of abstraction that works over a range of implementations that can be evolved and tuned over time, both within an application, and across applications.

Therefore, Croquet is based upon the concept of replicated computation – rather than replicated data. Croquet is based upon a synchronized message passing model, where the messages themselves ensure that the replicated systems remain consistent between machines. Though it is necessary to synchronize the world state of a new user by transferring the current contents of the world, after that, the worlds stay consistent only through the creation and processing of time based messages.

Implementation of TeaTime in Croquet 1.0

Croquet 1.0 does not implement "full TeaTime" replication semantics. That decision was made pragmatically because the expressive value of closures was not thought to be worth the complexity. Instead, two low-level and well crafted modifications have been implemented: "islands" and "routers". These are pragmatic solutions that are like the "stack-only" choice made by Lisp 1.5 (in LISP, a choice was made not to implement lambda-calculus closures - but instead to limit the LISP semantics to those that can be implemented on a stack machine based on so-called applicative-order evaluation - a pragmatic decision based on a belief that the expressive value of closures was not thought to be worth the complexity). We hope that eventually these will be modified to support the TeaTime full semantics. In the meanwhile, islands and routers meet the needs of our present efforts to deliver scalable multiuser virtual worlds without the need for reliance on centralized server infrastructures. Our design approach sets the stage for the full TeaTime semantics to be slipped back underneath Croquet with little impact at some time in the future.

Islands and Replicated Computation

The basic unit of replication and collaboration in Croquet is called an Island. A Croquet Island is first a secure container of arbitrary objects and access to the contents of an Island is governed by strict rules that ensure proper bottlenecking. Internally, objects see no difference between the rights they have in accessing other objects than in any other traditional system – except for a significant restriction that explicit infinite loops are simply not allowed. This is replaced instead by a mechanism we refer to as “temporal tail recursion” (discussed later). A single Island is actually made up of a collection of containers all in identical states on different machines connected by a network. One of these containers on a particular machine could be thought of as a replica or projection of the Island - albeit a complete projection. Croquet guarantees that the evolution of the state of a particular Island replica is identical to any other replica of the same Island. This is the basis of the Croquet collaboration architecture. Readers should take note that the term “Island” is used in several ways in this document. As just described, an Island consists of the set of all of its replicas. For clarity, we may refer to a copy as an “Island replica” (or simply “replica”), while the whole set is called the “Island”. Furthermore,

Island is a Squeak class that implements the base “Island” model; a replica is an instance of the Island class.

Croquet Islands

Croquet Islands are secure containers of other objects and are the basic unit of replication. They act as a kind of meta-object in that they have perhaps an even better model of encapsulation - certainly more secure, than traditional object models and they enforce a rigorous content-hiding and message passing model. This is a necessary precursor to guarantee identical behavior and identical response to external events (a similar concept to Islands is the Vats in the E programming language). Croquet Islands are generic object containers that are simple to checkpoint and exchange. They can easily be saved to disk for use later or archiving, or they can be transported between users to initiate collaborative interaction with the contents of the Islands. Objects internal to an Island have the same access privileges to each other that ordinary objects have. They can send messages directly to each other or themselves, can maintain direct access links to each other, and in general exhibit the same kinds of relationships that ordinary objects enjoy. They cannot, however, send messages outside the scope of the Island – nor can objects outside the Island send messages directly to the objects inside. That is not to say that there is no way for external messages to be sent to an object inside of an Island.

A TFarRef is an object that exists outside of the Island, but can act as a proxy for an object that is actually inside the Island. A message sent to the TFarRef is forwarded on to the actual object that is inside. This is actually a somewhat simplistic view of what actually happens – the actual process is a bit more interesting. The TFarRefs are actually generated by having the Island register a particular object as being externally accessible. An external name is generated, and a TFarRef is made available by the Island. The Island maintains a Dictionary that maps the TFarRef back to the original object. In Croquet, messages are sent to the original object inside of the Island indirectly via the TFarRef. This ensures that we have a nice way of bottlenecking the message, as we will usually have to redirect it in such a way that ensures it is properly replicated. Though there are ways to bypass this bottlenecking, it is extremely dangerous do so, as it can easily lead to a violation of the replicated state of the system. This invalidates the guarantee the Croquet has of ensuring perfectly replicated state inside of an Island. However, certain actions simply cannot be performed in a replicated way. An example of this is rendering of the content of an Island. Rendering only makes sense to a local observer, and is a relatively expensive action. Replicating the action of rendering a scene in Croquet is not only inefficient; it also does not make much sense.

Replicated Islands

Islands in Croquet are the units of replication in the system. For Islands to work properly, they must be deterministically equivalent. This means that given an identical initial state between two Island replicas, and given exactly the same inputs at the same time – the end states of these Island replicas must be identical. If for some reason there is even a slight divergence in state, this can easily be multiplied such that the end results are completely out of sync. Since the entire point of Croquet is to provide the users a perfectly replicated simulation environment that can be used as the basis of communication of information and ideas, this kind of breakdown renders the system totally useless. Of course, the entire point of the Island architecture is to have any number of Island replicas exhibiting identical state anywhere on the network – hence, anywhere in the world or even beyond.

A number of new concepts and objects need to be introduced to describe how the replicated Island architecture works. The first is the Croquet Message, which includes not just the token message name, but the target of the message, the message arguments and when the message will be executed. The second is the Croquet Router, which is the object that manages messages that are generated externally to an Island but are sent to it. It both determines when this external message will be executed and replicates it to all of the Island replicas. The third is the Croquet Controller, which is the interface between the Island and the Router and manages external events by redirecting them to the Router. Together with Islands, these are the main elements of a robust time-based replicated architecture.

Croquet Messages

A Croquet message is made up of four components, the target – which is the object that will actually execute the message, the actual message, the arguments to the message, if any, and the time at which the message will be executed. The time value is also used to sort the unexecuted message in the Island's message queue. Croquet messages can be generated either internally, as the result of the execution of a previous message inside of an Island, or externally, as the result of an external event usually generated by one of the users of the system. There is virtually no difference between internally and externally generated messages as far as the internal execution of the Island is concerned. A major difference between the two is that the timestamps on externally generated messages are used by an Island to indicate an upper bound to which the Island can compute its current message queue to without danger of computing beyond any possible pending messages.

Timing

The definition and manipulation of time plays the central role in how we are able to create and maintain a replicated Island state. We must be able to guarantee that every internally generated message will be executed in exactly the proper order at exactly the proper time. Externally generated messages must be properly interleaved with the internally generated messages at exactly the right time and order. When a new message is generated, it is inserted in the sorted queue based upon its execution time.

Island Time

An Island's view of time is defined only by the order of the messages it has in an internal queue. Islands can only respond to external, atomic, time-stamped messages. These messages are literally the Island clock. Though Islands have internal time based messages that can be queued up, these cannot be released for computation until an external time based message has been received which indicates the outer temporal bound to which the Island can compute to. This is a key point of the architecture. Though we may have a huge number of internal messages ready to be executed, they remain pending until an external time stamped message is received indicating that these internal messages are free to be computed up to and including the newly received message. Each Island's message queue is processed by a single thread, so issues with improperly interleaved messages do not arise.

When a message is executed, the time remains atomic in that it does not advance during the execution of this message. The "now" of the message stays the same. When we generate a future message during the current message, we always define its execution time in terms of the current "now" plus an offset value. This offset must always be greater than zero (though in fact zero is an acceptable value in certain circumstances, it should almost always be avoided because if it is infinitely iterated, Croquet can't advance and the system will appear to freeze.) If we generate multiple future messages, they will have an identical "now", though they may have different offsets. If we generate two messages at the same "now" and with an identical temporal offset value, an additional message number is used to ensure deterministic ordering of the messages.

All of the messages in the Island queue are "future" messages. That is, they are messages generated as the result of the execution of a previous internal message with a side effect of sending messages to another object at some predefined time in the future, or they are messages that are generated as the result of an external event – usually from a user, that is posted to the Island to execute at some point in the future, usually as soon as possible. All of these messages have time stamps associated with them. The internal messages have time stamps that are determined by the original time of the execution of the message that initially posted the message plus the programmer defined offset. The external messages have a time that is determined by an external object called a router and is set to a value that is usually closely aligned with an actual time, though it doesn't need to be.

Internal future messages are implicitly replicated; they involve messages generated and processed within each Island replica, so they involve no network traffic. This means that an Island's computations are, and must be,

deterministically equivalent on all replicas. As an example, any given external message received and executed inside of a group of replicated Islands must in turn generate exactly the same internal future messages that are in turn placed into the Islands' message queues. The resulting states of the replicated Islands after receipt of the external message must be identical, including the contents of the message queues.

External future messages are explicitly replicated. Of course external messages are generated outside of the scope of an Island, typically by one of the users of the system. The replication of external messages is handled by an object called a Router, which in addition specifies when the message will be executed. The Router is more fully described below.

External non-replicated messages are extremely dangerous and must be avoided. They do play a role, but it is extremely rare that anyone will ever have a need to make use of this mechanism. The problem is obviously that if a non-replicated message is executed and happens to modify the state of an Island it breaks the determinism the Island shares with the other replicated copies. We do use such non-replicated message when rendering the contents of an Island, but this is extremely well controlled.

Each Island has an independent view of time that has no relationship to any other Island (Island used here as the complete collection of Island replicas). This includes (for example) that a given Island could have a speed of time (relative to real time) that was a fraction of another. This is useful for collaborative debugging, where an Island can actually have a replicated single step followed by observation by the peers.

Since time is atomic and the external messages act as the actual clock, latency has no impact on ensuring that messages are properly replicated and global Island state is maintained. It does mean that higher latency users have a degraded feedback experience.

Islands enforce an internal "temporal tail-recursion" with the use of the #future: message. Basically, a message is arranged to execute some unit of time from the atomic "now" in the future. Hence, a message like:

```
#turn: angle
    cube rotateAroundY:angle.
    (self future:100)turn: angle+1.
```

causes the angle to be increased by one degree every 100 milliseconds.

The Croquet Router/Sequencer

The Croquet Router plays two major roles. First, it acts as the clock for the replicated Islands in that it determines when an external event will be executed. These external events are the only information an Island has about the actual passage of time, so the Island simply cannot execute any pending messages in its message queue until it receives one of these time-stamped external messages. The second critical role played by the Router is to forward any messages it receives from a particular Croquet Controller to all of the currently registered Islands. Given that Islands cannot execute beyond these external messages, it is usually necessary to manufacture new messages simply for the sake of moving time forward. These messages are created by the Router and are called *heartbeat messages*. They are basically message-free and contain only a time-stamp that allows the island to execute to. Routers can be located almost anywhere on the network and need not be collocated with a particular Island. Typically, the creator of the Island will own the Router by default.

The Croquet Controller

The Croquet Controller is the non-replicated part of the Island/Controller pair. The role of the Croquet Controller is to act as the interface between the Island and the Router and between the user and the Island. Its

main job is to ship messages around between the other parts of the system. The Controller also manages the Islands message queue, by determining when messages will get executed. Interestingly, a Croquet Controller can exist without an Island, acting as a proto-Island until the real island is either created or duplicated. In this case it is used to maintain the message queue until either a new Island is created or until an existing Island is replicated.

Croquet Message Execution

The basic idea behind Croquet's replicated message model is that the Croquet Router acts as the clock for all of the Island replicas. This is a guarantee that they all share exactly the same model of time. The Croquet Controller acts as the interface between the Router and the user and the Router and the Island. Every replica of an Island has its own Controller, but there is only one Router for the set of replicas of an Island.

To track a message from an initial event to execution inside of an Island, we first consider a user interacting with a specific object. The user never has direct access to the objects inside of an Island, so he can only interact with a far reference to that object, a TFarRef. A message is constructed using the following line of code:

```
farRef future aMessage:arguments
```

What we are doing here is sending aMessage: to the farRef to be executed as soon as possible in the future. In fact, another way to read #future is "ASAP". The farRef forwards this message to the Croquet Controller of the Island that contains the actual object that the farRef refers to. The Controller simply forwards the message again to the Croquet Router associated with the Island. The Router immediately places the current time stamp on the message – note that no two time stamps are equivalent – and forwards the message, now containing the execution time, back to the original controller. The controller then inserts the message into the Islands message queue and begins to execute all of the messages that are already in the queue that have a time stamp less than the new message.

This may seem a bit round about just to get a message to the local Island, but this process makes much more sense when seen in the context of replicated messages as described in the next section.

An interesting thing to point out is that a given internal message can generate a new message at a delta from the time of the original internal message that is actually less than the time of the new external message that is driving the clock. This newly minted message is simply added to the queue and executed in its own proper order before we actually execute the external message. If in turn its delta is small enough, it may even generate a number of additional messages that get executed before the external time stamped message is.

Replicated Message Execution

The main reason for the existence of the Islands, Routers, and Controllers is to enable the perfect replication of even complex interactions and simulations. The model for this is basically identical to the description in the previous section up to the point where the no time-stamped messages are sent out of the Croquet Router. A replicated Island has multiple identical copies of itself in various locations around the world (or the office or school). There is still only one Router, but now, for every Island replica there is a Croquet Controller.

After the Router receives a message from one of the Controllers, it now forwards the time stamped message to all of the Controllers connected to it – including the original one. The Controllers all insert the new message into the sorted message queues and executes the messages in each queue up to and including the new message. This means that every Island is now completely up to date with every other one.

Starting, Joining, and Participating

The process of creating a new Croquet session from scratch and then having new users join into the process is relatively simple. There are three parts to it – creating the Router, Controller, and Island – Joining the Router – and Participating in the Croquet session.

Starting Up

The first action required in creating a new Croquet world is for us to create a new Croquet Router. This Router can be on any accessible machine on the network – either remotely on a WAN, locally on the LAN, or on the same machine that will act as host to the original Island. Routers are extremely lightweight objects, so they really don't take up many resources, either in space or computation. The Router has a network address and port number that is how we will find it later.

Once the Router exists, we need to create a new Croquet Controller. This needs to be located on the same machine that the new Island will be located on, and is usually the original users own computer. Again, this is not essential. It is quite easy to create a Croquet Controller/Island pair on a remote server. Of course, this may take a few more resources than a simple Router requires. We give the Controller the address and port number for the original Router and it begins to connect.

Joining

The first thing the Controller does is send a message to the Router asking to subscribe to its message stream. Given that we made both the Router and the Controller, we are guaranteed of getting access – but it is important to note that this may not be true for other users as they attempt to join. You will have to grant them explicit permission – or leave the Router open to anyone, if they are to join the session. Once the Router authorizes the Controller it will begin publishing its message stream to it.

The only messages coming from the Router at this point are the heart beat messages – assuming we set the Router to generate these. In any case, the Controller is designed to simply begin adding these messages to its message queue. This is actually important when we are joining an already existent replicated Island, because in that case many of the messages that get sent and stored on the queue will be necessary to bring the Island replica up to date after it is replicated locally. Joining is view only access. At this point, even if there were an Island, the user is not allowed to send messages that might modify it in any way.

At this point, we can create a new Island from scratch using the Controller. We can also populate the Island and have the objects inside of it begin the process of sending their own internal messages into the message queue. Once the Island exists, we still need to be allowed to participate in it, which allows us to send it external messages generated by user events. Like joining, this is simply making a request to the Island to be allowed to participate. If granted, our Controller receives a list of facets, which is a kind of encrypted dictionary of messages that we are allowed to send from the user through the Router to the Island.

Adding Users

The new users need to be able to join a Croquet session while it is running, with minimal cost to the other users. If done properly, the other users might not even notice that another user has joined the session apart from seeing a new avatar appear in the scene.

Since a Croquet Router already exists, we only need to create a new Controller on our local machine. This is identical to creating the original Controller on the original users machine.

Just as before, the new Controller requests that it be allowed to join the ongoing Croquet session.

Once permission to join is granted by the Router, the new Controller will begin receiving messages from the Router. In this case, these messages will likely include events generated by the current users of the Croquet Island, so these messages are extremely important.

Now, instead of creating a new local Island, the Controller needs to request a copy of the current Island from the other user. The Router forwards this request to the original Controller and the Island is "checkpointed", in that a copy is made at a particular instant in time and streamed out to the new user via the Router.

Once the Island has been copied over to the new users machine – the message queue is truncated to the time of the most recent message executed by the Island, and execution seamlessly picks up from that point. The two users are now perfectly synchronized with identical Islands.

Participating

At this point, the new user has basically read-only access to the Island and must also ask to be allowed to participate. Again, it requests this from the Router and if granted the new user can begin to interact with his peers inside of the Island. The current system allows for both joining and participating with one message via the Croquet Harness.

Nice Side Effects

Because no messages are ever lost, and because the original message senders cannot specify when a message is to be executed, latency does not create timing or synchronization problems, just feedback problems. Systems will act sluggish if you have a higher latency, but the contents of the Island will remain identical between all users regardless.

This also means that users are not punished for having a high-latency participant sharing an Island, though the high-latency participant may have a less than satisfactory experience.

Since Routers are independent of Island/Controller pairs, they can be positioned anywhere on the network. This means that they can be moved to a position of minimal group latency, or onto centralized balanced latency servers. Routers can even be moved around if necessary to improve latency for specific users or groups for a certain time period.

The Future of Croquet Objects

The real work of Croquet is actually performed by the objects that are inside of the Islands. These are the objects that know how to display themselves, respond to external user events, and perform time-based simulations. They can be 3D objects that get rendered using OpenGL, or 2D objects that lie flat on the screen, or even zero-D objects that have no visual representation at all, but can perform complex computations.

In fact, there are really no special "Croquet" objects. The real distinction is that objects inside of Islands can send and receive future messages. These are virtually any message that an object understands, but sent into the future to be executed at an explicit later time. The syntax is basically the same as sending a normal message to an object, except we need to specify how far into the future the message will be executed.

As an example, if we want to rotate a cube in the 3D world by ten degrees around its y-axis we would normally write:

```
cube addRotationAroundY:10.
```

This would be executed immediately, and the cube would be rotated 10 degrees. If instead, we wanted to perform this operation sometime in the future, perhaps one second, we would write something like this:

```
(cube future:1000) addRotationAroundY:10.
```

The only real difference is the #future:1000, that specifies we want the next message - #addRotationAroundY: - to be executed 1000 milliseconds, or one second from now.

Virtually any object can be sent messages this way. Outside of an Island, we only have indirect references to the objects inside of it. We can still send messages to these inside objects via the reference, but we cannot specify when these messages are actually executed. To send a message to the cube inside of the Island, we first need to have a TFarRef to the cube – call it farCube, and to send the translation message we would do something like the following:

```
farCube future addRotationAroundY:10.
```

We cannot specify how far into the future this message will be executed. The only guarantee is that the Router will attempt to have it execute as soon as possible. If it is necessary to have a delayed future send, then you will need to write another method that in turn performs a future send of the required time. As an example, if you want to have the rotation triggered in five seconds after the external future send, you could write the following method:

```
MyCube>>#addRotationAroundY: angle when: time
    (self future: time) addRotationAroundY: angle.
```

Then, you would execute the following from outside of the Island:

```
farCube future addRotationAroundY:10 when: 5000
```

Note that this is still relatively undefined. All you know is that the actual add rotation message will be executed exactly five seconds after this message is executed, and all you know about that is it will be dependent on the best efforts of the Router and network.