Detecting Cheaters in a Distributed Multiplayer Game

Justin D. Weisz
School of Computer Science
Carnegie Mellon University
jweisz@cs.cmu.edu

Abstract

Cheating is currently a major problem in today's multiplayer games. One of the most popular types of cheating involves having the client software render information which is not in the player's current field of view. This type of cheating may allow a player to see their opponents through walls, or to see their opponents on a radar, or at extreme distances, when they would normally not be able to. Currently, much research is being done to learn how cheating can be detected and prevented in the context of client-server multiplayer games, but little research is being done studying how cheating behaviors can be detected or prevented in a distributed context.

In this research, we study what kinds of cheating behaviors are possible in a distributed game environment. First, we create an implementation of Mercury, a scalable publish-subscribe system for internet games, and we evaluate the implementation's performance. Second, we create a framework for interfacing a game with the Mercury system, and for managing distributed game state. Finally, we write a distributed game using this framework in order to learn what cheating behaviors are possible in this new distributed context. Several cheating behaviors are identified in this context, and several algorithms are developed to detect when players in the system are cheating. One of these algorithms is designed to detect discrepancies in a player's movement, and it's accuracy is measured. The accuracy of this algorithm is not yet adequate for use in a real-time game, and future enhancements to increase its accuracy are discussed.

1 Introduction

Multiplayer video games have become increasingly popular over the past decade, with the online gaming market predicted to be worth $2.3 billion by 2005 [17]. Services such as Microsoft's XBox Live [22] have introduced multiplayer gaming to the video game console, lowering the cost of participating in online games, and increasing the number people playing online games. Even movie theatres are being converted into large gaming centers, increasing the availability and exposure of online multiplayer games [15].

Current multiplayer games are divided into two types — first person shooter (FPS) games, which can have up to around 64 players in a game world [19], and massively multiplayer online role playing games (MMORPG), which support around 6,000 players in one world [17]. Both of these types of games are immensely popular. For example, the GameSpy network [7] is a service used by game players to find other game players to play games with. On a Sunday evening at 5:30pm EST, GameSpy reported having approximately 200,000[1] players using their system. Dark Ages of Camelot, one of the more popular MMORPGs, had on the order of 28,000 players playing on this same Sunday evening. Both of these statistics show that multiplayer online games are extremely popular, and played by a lot of people.

Because of the recent trend in increasing the number of simultaneous participants in a single online game, solutions have been developed to allow a game developer to run game servers supporting up to 32,000 players [25]. However, these solutions are proprietary, and require that a single entity (such as the game's developer) run the game server. This can be a problem when a game developer chooses to discontinue support for their game, leaving players out in the cold. Examples of this include multi-user shared dungeons (MUDs) which are no longer run by anyone, and hence are no longer playable (the original GemStone is one example of this).

[1] This number varies with the day and time; observed player counts have been observed from 70,000 to 200,000 at different times and on different days.

Therefore, the main motivations for this research are to scale real-time games to support hundreds of simultaneous players in order to keep up with industry trends, and to create a fully decentralized multiplayer game not dependent on any central authority. These motivations give us the following objectives:

- Design a distributed system upon which a real-time game can be built.
- Implement this system.
- Evaluate the performance of the implementation.

In developing a multiplayer game, there is one very important issue which arises, and that issue is cheating.

Cheating in online multiplayer games is a very big problem, and many people do not feel that they are doing wrong by cheating [20]. This mentality leads to the creation and wide-spread use of tools for cheating in games. For example, there are programs for the game The Sims Online which will play automatically, in order to increase the status and wealth of a player's character. This is normally a time-consuming process, but having automated software automatically play the game frees one up from the tedium of actually playing the game, in order to be highly ranked among other players. Ironically, players who use these tools do not feel they are doing anything wrong, but rather feel that they are making the game more fair, because they are not actually able to devote the same amount of time to the game as other players [20].

The use of automated software also takes place in other games as well. For example, in Diablo II, automated software can be used to find rare items, and duplication bugs exist which allow a player to duplicate any item they wish. In the popular game EverQuest, which boasts the 77th most wealthy economy in the world [10], entire characters have been sold for upwards of $\$1,400$[2]. Because of this widespread selling of EverQuest items and characters, Sony has had such sales banned, in order to keep the game fair for honest players [16].

All of these examples illustrate how multiplayer games can become unbalanced, because a population of users are able to perform behaviors in the system which give them an unfair advantage. In multiplayer games, the actions of one user have a direct consequence on the actions of other users, and when a user behaves in such a way as to unbalance the game, it can ruin the enjoyment of the game for others. This is especially true if one player's actions gives him the power to completely dominate over his opponents. Or worse yet, having tangible rewards for performing activities not intended by the game's developer, such as duplicating items, can encourage players to perform these activities, further upsetting the game's balance.

Returning the discussion to FPS games, such as CounterStrike or Quake III, automated software exists which gives players an extremely unfair advantage over their opponents. These programs are called "aim bots", and they automatically aim a player's targeting reticle over their opponent's head [14]. This allows the player to kill their opponent without having to aim manually, and without needing any real skill in the game [2].

Cheating detection has been studied in FPS games, and software has been developed to detect when a player is cheating. PunkBuster, created by Even Balance, Inc. is one of the more popular anti-cheating programs, and has been incorporated into many of today's popular FPS games [12]. PunkBuster works by scanning the memory of a player's computer for known cheats and exploits, and reporting them to the game server. However, this system is dependent on a client-server architecture, where the server is a trusted entity, and as such, cannot be used in a distributed environment.

Ultimately, behavior on the Internet is characterized by reduced inhibition [8]. People are less inhibited in their speech and actions, and thus are more inclined to behave in a way which is more deviant from traditional social norms. Our research is concerned with deviant behavior that impacts negatively others. In a general context, our process is one of identifying behaviors which can cause a negative impact, and of create a method to detect when these behaviors are occurring.

Therefore, in the specific context of a distributed multiplayer game environment, and after satisfying the objectives presented above, our goals are to:

- Examine what cheating behaviors can be executed in the environment,
- Create an algorithm to detect these behaviors, and
- Evaluate the performance of this algorithm.

In the next section, we give an overview of the project, describing each component of the implementation and of the multiplayer game. In Section 3,

[2]These are real US dollars.

we explain MERCURY, a scalable publish-subscribe system specifically for multiplayer games. Sections 4 through 6 detail our implementation of MERCURY, the framework we created for interfacing MERCURY with an application, and the multiplayer game we developed. In Section 7 we describe potential cheating behaviors which could be executed in the system, present an algorithm to detect when those behaviors are being used, and evaluate the performance of this algorithm. Sections 8 and 9 detail future research directions and our major conclusions.

2 Project Overview

Our first objective in this research was to find a routing protocol which allows for the distribution of game state among the nodes participating in a game, without the requirement of a central server or authority. This protocol must also be able to handle game state updates at a fast enough rate (i.e. the protocol implementation must not add a significant amount of overhead) in order to meet the requirements of a real-time FPS game, such as Quake[3].

The protocol we chose to use in our project is called MERCURY, which is a fully distributed publish-subscribe routing protocol for Internet games. MERCURY has been documented in [3], and uses a novel content-based routing protocol. The reason we chose MERCURY is for its subscription language, which allows a node to receive publications based on the contents of those publications. The details of MERCURY, as well as a primer on publish-subscribe systems, are presented in the next section.

Next, we created an implementation of MERCURY in C++, since it has only previously been implemented in a simulation environment. We also evaluated it's performance, using the Emulab network testbed [4]. Details of the MERCURY implementation, as well as it's performance evaluation, can be found in Section 4.

Finally, we sought and found a game environment for studying cheating and cheating detection, and created several algorithms for detecting these behaviors. The behaviors we were interested in were behaviors that either exposed more information to a player than they were privy to, or behaviors which altered the game state in a way inconsistent with the normal course of game events. For example, altering one's movements contrary to the rules prescribed by the game's physics was a cheating behavior we were profoundly interested in. In order to study and focus on cheating behaviors and methodologies, the gaming environment we created was rather simplistic in nature, designed with as few specific implementation details of game programming as possible. This environment is discussed in Section 6, and an analysis of the cheating behaviors we discovered, as well as algorithms to detect them, can be found in Section 7.

[3]The ideal "ping", or amount of time for a packet of information to travel from the player's computer to the server and back, is less than 50ms. Therefore, the protocol overhead should not be significantly larger than this number.

2.1 Project Structure

The environment we developed for studying cheating detection can be seen in Figure 1 as a three tier system. On the bottommost layer lies the routing protocol, which is currently our implementation of MERCURY. One feature of our design is that other routing protocols can be implemented and swapped in, in place of MERCURY. This allows us to compare the performance of several different publish-subscribe systems, all in the same context. In order to accommodate this "hot-swappability", the framework layer was added to serve as an intermediary between the application layer and the routing layer. The framework is responsible for the management of game state, for publishing world objects when they have changed their state, and for providing a clean interface between the application and routing layers.

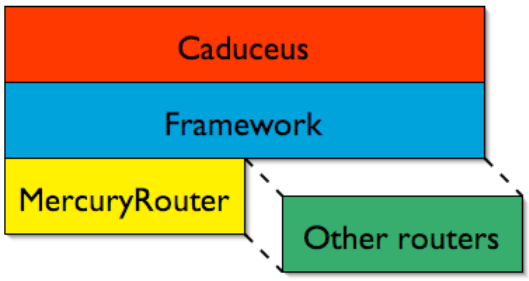

Figure 1: System component diagram

Once the implementation of MERCURY and the framework were complete, we created an application built on top of this foundation, named CADUCEUS. This application is a real-time two-dimensional space shoot-em-up game, based off of the popular game "KAsteroids" [9], which is in turn based off of the classic arcade game "Asteroids".

Now that we have given an overview of the components of this system, we next present a description of

the MERCURY routing protocol.

3 Mercury – Theory

A publish-subscribe system such as MERCURY can be viewed in many different lights; it can be seen as a distributed database, or as a method of content delivery. Both of these views are applicable to our project. For both of these views, we define the notion of an *object* as an encapsulated piece of information which represents a literal object in the game world. In a game such as Quake, information regarding a player's health, location and status would be encapsulated into a player *object*. The various pieces of information which compose an object are known as the *state* of that object. There is a special type of state known as an *attribute*, which is used as part of MERCURY's content-based routing scheme.

In a publish-subscribe system, there are two types of participants: publishers and subscribers. For the purposes of this example, we consider these two parties to be separate entities, but they can be one and a same. In the context of a content delivery system, a publisher is a *content-generator*, and in the context of a distributed database, he is a *database updater*. The publisher lets others in the system know when an *object* has changed its state, when a new object has entered into existence, or when an object has ceased to exist. A subscriber, on the other hand, is interested in receiving certain objects when they are published. The specific criteria for receiving these objects is up to the subscriber, and is based on the values of the *attributes* of an object. For example, if the objects in the world are pieces of text, a subscriber can choose to receive all objects which have a "font" attribute of "bold", or a "color" attribute of "red". The MERCURY routing system's main function is to match publications with subscriptions, in order to deliver relevant publications to those who subscribed to them.

3.1 Publications and Subscriptions

To understand how the routing system works, the concept of a *named, typed attribute* must first be understood. In a publish-subscribe system, publishers publish objects which are collections of state and attributes, and each of these attributes has three components: an *attribute name*, a *type*, and a *value*. The *attribute name* is what the subscriber uses as their criteria for receiving publications. The *type* represents what kind of attribute the attribute is, and can be one of {`char`, `int`, `float`, `double`, `string`}. These types correspond to the identically named standard C primitive types, with the exception of `string`, which is the C++ "string" type. Finally, the *value* is the actual value of the attribute, and is of the type specified in *type*.

Now that the building blocks of a publication are known, here is an example of a publication, for a very simple object which represents a player in a 2D game:

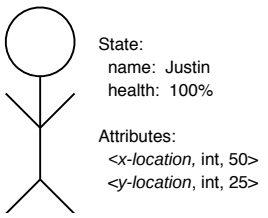

Figure 2: Example of a publication in a simple 2D game

In order for my friend, who is also participating in this simple 2D game, to receive my player object when I publish it, he needs to create a *subscription*. The MERCURY subscription language is a subset of a relational SQL-like query language, and allows for rich and complex queries. A subscription is a conjunction of fields of the form: {*type*, *attribute name*, *operator*, *value*}. The *attribute name*, *type* and *value* fields have identical definitions as for publications. New to subscriptions is the *operator* field, which can be one of $\{ <, >, \leq, \geq$ and $= \}$. In the case that the *type* is a `string`, the string operators *equal*, *prefix* and *postfix* are also valid. All operators have their implied meaning, and in the case of strings, $prefix(A, B)$ means that string B has as it's first $length(A)$ characters the string A. $Postfix(A, B)$ means that the string B has as it's last $length(A)$ characters the string A.

To continue the example of a 2D game, one subscription which contains the publication shown in Figure 2 would be as follows[4]:

3.2 Routing Mechanism

In order to satisfy the needs of a real-time multiplayer game, the routing mechanism must be both fast and efficient, as well as correct. By correct, we mean that publications must be delivered to those

[4]There are many other specific subscriptions which would "cover" the publication; this is only one such example.

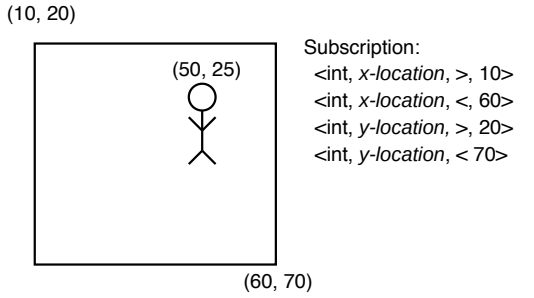

Figure 3: Example of a subscription in a simple 2D game

parties who subscribed to them — publications cannot be dropped for the sake of efficiency[5].

Attribute Hubs

MERCURY divides publication and subscription routing responsibility among nodes by partitioning them into groups called *attribute hubs*. Each hub is responsible for a single attribute in the overall schema, where the schema is the entire list of named, typed attributes in the system. Let H_a denote the attribute hub for attribute a.

Nodes inside each attribute hub are logically arranged in a circle, with every node keeping track of its successor and predecessor. Each node is responsible for a *range* of attribute values. For numeric types, the mapping is straightforward since they have a bounded range; at worst, it would be bounded by the size of the data type. Thus, for an attribute a of integer type (32 bits), each node would be responsible for an interval of length $2^{32}/|H_a|$. In practice, the dynamic range may be much smaller. For example, in Figure 4, the maximum value of either coordinate is dictated by the size of the virtual world.

String attributes, on the other hand, can have arbitrary length, so partitioning them efficiently requires making a few trade-offs. By partitioning them on the basis of first few or last few characters, we can support either prefix or postfix operators efficiently. For example, a node can be in charge of all strings starting with '`d-f`', and then the subscriptions stored at this node can match against publications having a certain string prefix. Because of this, the *substring* operator cannot be efficiently supported by the MERCURY system. This demonstrates the trade-off between scalability and expressiveness of the selectivity mechanism. In our implementation of MERCURY, we chose to support string prefixing rather than postfixing.

[5]Dropping publications may be an improvement that can be made in the future, but since this system is being developed with more than just video games in mind, we currently seek to ensure correctness.

Routing of Publications and Subscriptions[6]

Let $\mathcal{A}$ denote the set of attributes in the namespace. Let $n = |\mathcal{A}|$ be the cardinality of the set. For example, in a FPS game, $\mathcal{A}$ could be { `x-coordinate`, `y-coordinate`, `z-coordinate`, `event-type`, `player_name`, `team` }. Suppose that $\mathcal{A}_S$ denotes the set of attributes in a subscription S. Similarly, let the set of attributes present in a publication P be denoted by $\mathcal{A}_P$.

The routing of subscriptions and publications is done in the following manner: A subscription S is routed to H_a, where a is *any* attribute chosen from $\mathcal{A}_S$.

Given a publication P, the set of subscriptions which could match P can reside in any of the attribute hubs H_b where $b \in \mathcal{A}_P$. Hence the publication P is sent to *all* such H_bs. Thus, it is possible that a publication could be sent to all the n attribute hubs. However, we believe that in a typical application, only a few attributes will be popular, such that most subscriptions will contain reference to at least one of the popular attributes. In this case, hubs for these attributes alone will suffice. In the rare case of a subscription not containing any of these attributes, we can send it to all the hubs.

In the most naïve routing scheme, content gets routed along the circle using successor and predecessor pointers. A node in H_a compares the value of attribute a in a publication or a subscription to the range it is responsible for. If the value of a falls within the node's range, then that node is said to be the *rendezvous point* for that publication or subscription. Depending on the results of the comparison, it stores and/or forwards the message appropriately.

Figure 4 illustrates the routing of subscriptions and publications. It depicts two hubs H_x and H_y corresponding to the X and Y coordinates of a player. The minimum and maximum values for the x and y attributes are 0 and 320 respectively. Accordingly, the ranges are distributed to various nodes. The subscription enters H_x at node `d` and gets stored at nodes `b` and `c`. The publication is sent to both H_x and H_y.

[6]Portions of this section are reproduced from [3] with permission.

In H_x, it gets routed to node `b` and matches the earlier subscription, while in H_y, it is thrown away after getting routed to node `e`.

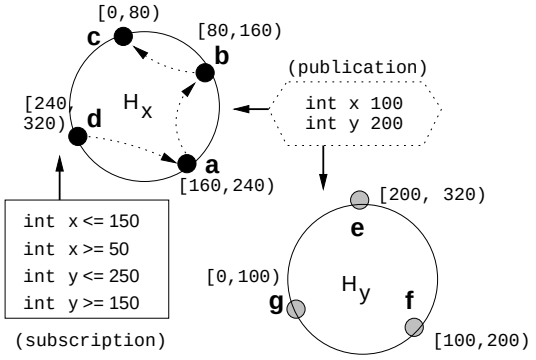

Figure 4: Routing of publications and subscriptions. A subscription is routed to any *one* of the attribute hubs H_x or H_y. In the figure, the subscription is stored at nodes c and d. The publication is routed to *both* hubs since any one of them could have stored the relevant subscriptions.

In this scheme, a publication gets routed to *exactly* one rendezvous point since it carries a *single* attribute value. A subscription, however, can contain ranges, and hence, can be stored at any number of rendezvous points depending on how much attribute space it covers.

One final feature of the routing protocol which needs to be discussed is how a node joins the system. In order to be able to direct a node to the appropriate attribute hub, some subset of nodes needs to collectively know the entire attribute schema. We call these nodes *bootstrap nodes*, and they are the primary source of contact for a node who wishes to join the system. Learning the address of a bootstrap node can be done in an out-of-band fashion, and a node only needs to contact a single bootstrap node in order to join an attribute hub.

When a node J contacts a bootstrap node to join the system, the bootstrap node responds with all of the information required to connect to another node N within a specifically chosen attribute hub, along with the specific information about the attribute (absolute minimum and maximum value, attribute name, etc.). Load balancing can be performed at the bootstrap nodes by biasing how the attribute space is partitioned, in order to accommodate areas of the attribute space which are more or less popular. After J receives the information about N, J contacts N and sends a request to join. N then splits his attribute space in half, keeps the upper half, and sends the lower half to J. Then, N sends a message to his predecessor, informing him of the change in topology, and N, N's predecessor, and J update their predecessor and successor pointers accordingly.

Routing Optimization

There is a routing optimization we consider to enhance the performance of the routing protocol. In the above routing scheme, a publication takes on average $n/2$ hops along the circle to get routed to its rendezvous point. This average can be reduced by having a rendezvous point send an acknowledgement (ACK) whenever it matches a publication. This ACK contains the node range of the rendezvous point, and is sent to the node which originated the publication. When an ACK is received from a rendezvous point, the rendezvous point's address is cached, such that future publications can be sent directly to the rendezvous point, without having to travel a great distance each time.

4 Mercury – Implementation

This section describes our implementation of Mercury, and details the process by which packets are routed. Mercury has previously been implemented in a simulation environment, but has not yet been implemented as an actual computer program. Our implementation of Mercury uses the C++ language, with a very small section of C code used for debugging purposes. It has been compiled and run successfully on both Red Hat Linux 7.1 (Seawolf) and 8.0 (Psyche).

One notable feature of the implementation is that it is not dependent on the specific routing mechanism being used. Thus, many of the C++ classes we created can be re-used for a different publish-subscribe routing scheme, and some of the classes can even be re-used in other general networked applications.

A description of the network protocol is provided, followed by documentation of the C++ classes created for the implementation. Sections 4.3 through 4.5 discuss the specific process of joining an attribute hub and routing packets, and Section 4.8 discuss how well our implementation performs in a real-world benchmark.

4.1 Network Protocol

The MERCURY network protocol is built on top of the TCP/IP protocol stack to ensure reliable, in-order delivery of packets. The UDP protocol is used for sending periodic heartbeats in order to maintain network connections and send status information. Each network message is prefixed by 3 bytes of header data — one byte is used for determining what kind of message is being transmitted, and 2 bytes are used to determine how long the rest of the message is, in bytes.

There are 12 different kinds of messages which can be sent, and they are:

- `INVALID_MSG`: used to signify that an invalid message was read (such as when a connection is dropped prematurely)
- `MERC_HEARTBEAT`: heartbeat packets are sent periodically to verify that connections are still active, and to update the information at the bootstrap nodes
- `MERC_ACK`: when caching is being used, acknowledgements are sent whenever a publication is matched at a rendezvous point
- `MERC_JOIN_REQUEST`: this type of packet is sent when a new node wants to join an existing node in some attribute hub
- `MERC_JOIN_RESPONSE`: when a node receives a join request, this message is sent in response to let the node know of either the success or failure of the join
- `MERC_UPDATE_SUCCESSOR`: when a node joins an attribute hub, the node which is being joined needs to inform his predecessor that a new node is joining in between, and should update his successor to become the new node
- `MERC_CHANGING_SUCCESSOR`: when a node receives an update successor message, a changing successor message is sent to the new successor node
- `MERC_PUB`: this represents a publication of a world object
- `MERC_MATCHED_PUB`: when a publication is matched against a subscription, the publication is delivered to the subscriber as a matched publication, so the subscriber knows to deliver it to the application
- `MERC_SUB`: this represents a subscription to any number of world objects based on their attributes
- `MERC_UNSUB`: this represents an "unsubscription"; when this type of packet is received, the receiver removes the given subscription from his database
- `MERC_BOOTSTRAP`: this message is sent by the bootstrap server to a joining node, and contains information about the attribute hub and host the node is to join

4.2 Mercury Class Structure

There are six different types of classes used in the implementation, and they are: data, network, routing, utility, cache and byte conversion.

Data

The data classes are responsible for encapsulating and storing the actual data which is sent over the network. Each data class inherits from the same superclass, the Application Data Unit (ADU). The `ADU` class keeps track of what type of data is being stored by the particular class instance. The classes which inherit from `ADU` are:

- `AckData`: When a publication reaches its rendezvous point, an acknowledgement is sent back to the originator of the publication. This acknowledgement contains the address of the rendezvous point, as well as the range of attribute space it is responsible for (its `NodeRange`).
- `BootstrapData`: This class is used by the bootstrap server to tell a joining node the name of the attribute hub they are joining, the type of the attribute, the address of the host to join, and the absolute minimum and maximum values of the attribute.
- `HeartbeatData`: Each node in the system sends periodic heartbeats to the bootstrap server. These heartbeats contain a node's `NodeRange`, which allow the bootstrap server to keep track of how much attribute space is covered by each node. This information can be used in the future to help the bootstrap server load-balance attribute hubs when responding to join requests.

- `PubsubData`: This class is used to encapsulate both publication and subscription data, by storing pointers to the publication and subscription wrapper classes (see Section 5.1 for more information about these wrapper classes). The sender of a publication/subscription is also stored in this class.

There are also four data classes which are used when a node joins an attribute hub:

- `JoinRequestData`: When a node wishes to join another node, they send a join request to that node. This join request simply contains the address (IP and port) of the joining node, and is explicitly provided to work around firewall and network address translation (NAT) systems.
- `JoinResponseData`: After a node receives a join request, it responds with a join response. This response has an error flag signifying whether the join was successful or not. If a node tries to join another node which is not joined, then the error flag states this condition. The join response also contains the range of attribute space the joining node will be responsible for, as well as the address of the node who is to become the joining node's predecessor.
- `UpdateSuccessorData`: When a node successfully joins another node, it is fit in between two nodes in an attribute hub's circular topology. Thus, in order to keep the structure circular, a node sends an update successor message to its predecessor, after it has received a join request. This update successor message contains the address of the joining node, and upon receipt, the receiving node will update its successor to the node specified in the message.
- `ChangingSuccessorData`: After a node receives an update successor message, it sends a changing successor message to the new successor node. This lets the new successor node know that it has completed its joining process, and is now part of the attribute hub. This class does not have any data members, and is just used to signal the joining node that the joining process has completed.

A full example of the joining process is given in Section 4.4.

Finally, there is one data class which is responsible for storing all of the information pertaining to an attribute hub, and to a node's specific role within that attribute hub. That class is:

- `NodeRange`: Stores the name of an attribute, the type of that attribute, the minimum and maximum values that a node is responsible for, as well as the absolute minimum and maximum values the attribute can take. Each node in an attribute hub has a node range which covers some portion of that hub's attribute space.

Network

There are three classes which support network operations in our implementation. These classes are used to encapsulate network messages and relevant details of these messages, as well as manage TCP/IP connections and store information about remote hosts.

- `Message`: This class is a general-purpose "packet" class, which stores information about transmitted messages. The class has a pointer to an `ADU`, which stores the actual transmitted data. The sender of the message is also stored here, as well as the number of hops this message travelled to reach the current node. Finally, this class stores a timestamp of when the class was last serialized. This allows our benchmarking utility, discussed in Section 4.8, to track how long a packet takes to reach its destination.
- `Connection`: Connections are the most important piece in a networked application, and this class contains all of the necessary information to establish and maintain a TCP/IP connection. The socket, IP address, and port are all stored in this class, as well as information regarding the number of times connection establishment should be tried before failing. This class provides a member function which performs connection establishment, retrying as necessary.
- `Peer`: The peer class is used to keep track of other nodes in the system. Associated with a peer are the `NodeRange` it is responsible for, the IP address and port of the peer, as well as a `Connection` which can be used for communicating with the peer. Other information stored about peers includes a flag indicating whether or not they have joined an attribute hub, the timestamp of the last heartbeat received from the peer, and the time at which they joined the attribute hub.

Routing

As shown in Figure 1, the foundation of our system is the MERCURY routing layer. In order to provide an architecture where multiple types of routing schemes can be used, we created an abstract `Router` class which can be subclassed and implemented in order to support different routing policies. This class provides a clean interface to the framework layer, so the framework layer does not need to know about the specifics of the routing policy being used. There are currently two classes which inherit from `Router`, and they are:

- `FloodRouter`: This is a very simple router which floods all publications to every other node in the system. This router was developed primarily as a means of developing the `Router` interface.

- `MercuryRouter`: This is where the heart of MERCURY lies. This class is responsible for reading in packets from the network, forwarding them to peers, and delivering them to the framework if necessary. This class also reads in publications and subscriptions from the framework layer, serializes[7] them for network transport, and routes them as appropriate. The current implementation of `MercuryRouter` supports single attribute routing, as well as caching. Multiple attribute routing has also been implemented, but has not been used for our benchmarks or game.

  As for data members, this class keeps track of a lot of state. The router knows the address and port it is running on, it keeps track of the socket being used for listening for new connections, it stores pointers to its predecessor and successor nodes, it knows when it has joined an attribute hub, it keeps track of all of the active TCP/IP connections, as well as its connection to the bootstrap server for sending heartbeats. Subscriptions are also stored in the router, and are checked whenever a publication is matched.

Utility

A few utility classes were created to ease certain programming tasks. None of these classes are instantiated; rather, they are used to bundle together functions which perform similar tasks. These classes are:

- `BufferManager`: To avoid having to constantly allocate and deallocate buffers when reading from a socket, this class was created to manage a temporary buffer for this purpose.

- `UThreadFactory`: This class provides static wrapper methods for some of the common pthread functions — creating and reaping threads, as well as initializing, locking, unlocking and destroying mutexes. These wrapper functions check for any error conditions which may occur when using these functions.

- `UNetworkFactory`: This class provides wrapper functions for reading and writing data using sockets. These wrapper functions check for any error conditions returned when reading or writing.

- `Utils`: Miscellaneous functions, such as getting the current time, and taking the difference between two times, are stored in this class. Converting between an IP address in integer format to string format is also part of this class.

Cache

Caching is a major routing optimization we implemented, and as such, a few classes were created to support different types of caches. Much as with the `Router` class, the `Cache` class provides an interface for working with a general cache, and this class can be subsclassed in order to produce different types of caches with different caching policies. The caching classes are:

- `Cache`: Provides an interface from which specific types of caches can inherit, and stores how many entries are to be stored in the cache.

- `CacheEntry`: Each entry in the cache is composed of a node's address, its `NodeRange`, and the timestamp of when the entry was last accessed in the cache.

- `SingleCache`: This class inherits from `Cache` and provides a very simple queue-style cache, with new entries pushed to the back, and old entries popped from the front.

- `LRUCache`: This class implements an LRU style cache. New entries are inserted into the cache until it is full, and then the least recently used cache entry is evicted in order to make room for a new entry.

[7] The terms *serialize* and *deserialize* are synonymous with the terms *marshall* and *unmarshall*. In this paper, we will remain consistent and use the former two terms.

Byte Conversion

In order to ensure byte-ordering compatibility with MERCURY nodes running on different platforms, the byte conversion classes provide a mechanism for converting binary data (such as integers and floats) into network byte order for transmission over the network, and for converting binary data back to host byte order when it is received. The two classes which perform these functions are:

- `Swapper`: Provides an interface for a byte-swapping class used to convert binary data to and from network byte order. The three types of data which can be converted are `short` (2 bytes), `int` (4 bytes), and `float` (4 bytes). The actual class which is used for byte swapping is determined at runtime, based on a test of what byte-ordering scheme the host uses, little-endian or big-endian.
- `BigEndianSwapper`: This class is used on big-endian machines.
- `LittleEndianSwapper`: This class is used on little-endian machines.
- `ByteConverter`: This class is the front-end to the byte conversion service, and uses one of the swapper subclasses to perform the actual byte conversion. Primitive types can be either written to a temporary buffer or read from a given buffer, with all of the byte swapping taking place behind the scenes. This class is used whenever a `short`, `integer`, or `float` must be written to or read from the network (as is the case when reading and writing packet headers).

4.3 Bootstrap Server

The bootstrap server is the first point of contact when a new node wants to join an existing attribute hub. Because of the specific network topology which needs to be maintained when nodes join and leave an attribute hub, having one server in charge of managing the network topology is the easiest way to enforce the topology. The bootstrap server is implemented as a separate program, but can be run concurrently with an instance of a MERCURY node. An instance of the bootstrap server can handle any number of attributes, and in the current implementation, we use only one bootstrap server instance for all attributes in the system. Multiple bootstrap servers could be used in the future, with each responsible for a different set of attributes. This would also allow for a more dynamic attribute space, where the entire schema of the network is not completely known to any one node.

The bootstrap process is very straightforward — when a node wants to join the system, they connect to the bootstrap server. Then, the bootstrap server picks an attribute hub for the node to join, and tells the node who to connect to in that hub. In the current implementation, the bootstrap server keeps a fixed-size FIFO queue of "current" nodes in each attribute hub, where "current" is defined as having recently received a heartbeat message from a node.

4.4 Joining an Attribute Hub

After a node receives the address of another node to connect to, the joining process begins. This process is detailed in Figure 5.

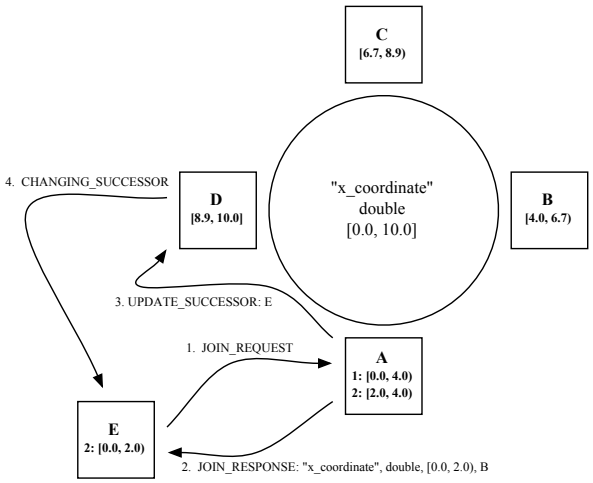

Figure 5: The joining process

In Figure 5, node E joins node A. At step 1, node E sends a `JOIN_REQUEST` packet to node A. At step 2, node A splits his `NodeRange` in half, and gives the lower range to node E. Since "x_coordinate" is of type `double` (i.e. a continuous type), node E's range becomes $[0.0, 2.0)$, and node A's range becomes $[2.0, 4.0)$. Note the open upper interval for all nodes in the attribute hub except for D, which must keep a closed upper interval because it is the attribute's absolute maximum value.

Once A has sent the join response to node E, he sends an `UPDATE_SUCCESSOR` message to node D, informing him that node E will become his new suc-

cessor. Node A also updates his predecessor link to point to node E, instead of node D. Node D now updates his successor link to point to node E, and he informs node E that his link has been updated by sending him a `CHANGING_SUCCESSOR` message. After this message is received by E, the join is complete.

4.5 Routing Packets

Routing packets works exactly as described in Section 3.2. There is one slight departure from the routing protocol which we make in our router, which deals with delivering packets to subscribers. When a publication matching our node range is received, we ask the framework if the application wants to handle the publication, or if we should deliver the publication to those who subscribed to it. The framework in turn asks the application what should be done, and the application then makes a decision. It can either:

- tell the router to forward the matched publication to subscribers, which is the normal routing behavior, or,
- drop the publication entirely, in which case the subscribers will not receive the publication, or
- drop the publication which was received, but create new publications which will be delivered to subscribers

This last behavior is the most interesting one, and is the one we perform in CADUCEUS. Briefly, the reasons for choosing this third path are because the objects which are published by the players in the game are not the same objects that get delivered to the subscribers; rather, the published objects act as *modifiers* for other objects which are stored at the rendezvous point, and it is these other objects which get delivered to the subscribers when a publication is received. More information about this design decision can be found in Section 6.2.

4.6 Debugging Features

In order to vie for the correctness of our implementation, as well as just figure out how the system is operating, we implemented a small suite of debugging aids.

Local Logging

Each instance of MERCURY creates a log file on startup. This log file is a dumping ground for all sorts of debugging information. For example, when a packet is received, output is written to the log file detailing what kind of packet it was, and what its contents were. Or, when a node splits its range as part of the joining process, the details of the split are recorded. The details of matching publications against subscriptions are also logged.

The treatment given to this feature may seem lacking, but it is worth mentioning because the local application log has been extremely useful in debugging and correcting our implementation. Having a separate place to store the actual contents of packets, rather than just dumping them all to the console, is very useful, especially when those contents are not always needed.

Remote Logging

In contrast to this, the more exciting logging feature is the remote logger. The remote logger is a separate program which reads in logging messages on a UDP port, and outputs their contents to a single log file. The log file is set up so that messages from different hosts are segregated and separately labelled, and the master log file provides a global view of the events that occur in the system, in order. When debugging the joining process, for example, the order of events shown in Figure 5 is clearly displayed in the master log, along with additional information about the details of the join. It should be noted that the remote logger is a tool which is not specific to MERCURY, and can be re-used for other applications.

4.7 Object Storage and Serialization

Now that the specific details of routing messages have been discussed, it is time to discuss what these messages are, and how these messages correspond to objects in some virtual game world. One of the major difficulties in programming a system to send structured data over a network is figuring out a good method for encoding that structured data. For example, a C++ class can contain primitive data, pointers to other objects, C style structures, and unions. During the initial phases of implementation, it was decided that objects would be sent over the network as a stream of these structures; so if a class was composed of an integer, a character, and a pointer, the integer, character and pointer would be sent over the network.

However, this leads to an obvious problem. Pointers are just memory addresses, and a pointer which

points to some object O on one computer may point to garbage on another. Therefore, our goal was to figure out how to serialize complex composite objects in order to transmit them over the network in a meaningful way.

Note that the term "object" is being used in an overloaded fashion. From the perspective of the game world, an "object" is some entity in that world. Using a previous example, a player in a game would be an "object" in the world. This object can have one of several *representations*, and they can be:

- an object stored in the computer's memory. This kind of representation entails that some C++ object has been instantiated in computer memory. There can be many different instantiations of the same object, spread across different computers. For example, for a player in a virtual world, each participant in the virtual world will have allocated memory to store this player object. However, there is really only **one** player object in the game world.
- an object stored in the computer's memory, in serialized form. While this representation is stored in computer memory, it is not stored in a format suitable for modification. Rather, it is stored in a format suitable for network transport, or for outputting to the console or a log file.

Now that this distinction has been made, the goal now becomes being able to convert between the memory representation to the serialized representation such that the serialized representation can be transmitted over the network to another node, and deserialized back into the memory representation.

In order to achieve this goal, we chose to use the extensible markup language (XML) [5] for the serialized representation of world objects. XML is a markup language for documents containing structured information, and as such, it provides a nice mechanism for representing both simple and complex C++ classes. The pointer problem is also solved using XML, because an object A containing pointers to objects B and C can serialize objects B and C as part of the serialization of object A.

To give an example of what a serialized object looks like in XML format, return to the example of a publication given in Figure 2. A serialized version of this publication is:

The example above is only one way to serialize an object, and is in fact much simpler than how objects are really serialized in our implementation.

```
<player>
  <name value="Justin">
  <health value="100">
  <attribute>
    <name>x_location</name>
    <typedval type="INT" value="50">
    <name>y_location</name>
    <typedval type="INT" value="25">
  </attribute>
</player>
```

Figure 6: Example of a serialized object

In order to perform actual XML serialization and deserialization, we use the Xerces C++ XML Parser [23] created by the Apache XML Project [24]. Xerces-C++ is a validating XML parser written in C++, and is fully open source. A wrapper class for interfacing with the parser was created, and provides parsing services to both the framework and routing layers.

4.8 Benchmarking Utility and Results

In order to test the speed of our implementation of MERCURY, as well as correctness, we ran benchmarks on the Emulab network testbed [4]. Emulab is a network of computers designed to emulate a real network, and allows researchers around the world to run network experiments in a very precise, controlled manner. Every aspect of an experiment on Emulab is customizable, from the network topology and link delays to the operating system each node runs. Thus, experiments performed on Emulab reflect the actual performance of a system, independent of any actual network conditions which may affect results.

For our experiments, we used 30 physical Emulab nodes with a randomly generated network topology. All of the link delays were set to 0 in order to ensure that only the routing and serialization overheads were being accounted for.

The experiments we performed involved setting up a single attribute hub for a `float` value between 0.0 and 1.0, and having a single node send publications of this attribute chosen from two different probability distributions: uniform, and Zipf. The uniform distribution chooses values in the range $[0.0, 1.0]$, with each value having the same probability of being chosen. The Zipf distribution, on the other hand, chooses values based on a power law. The Zipf distribution

Node count	Distribution	Cache	Avg. delay to match (ms)	Avg. hops
50	Zipf	no cache	249	18.358
		$\log n$	42	3
	Uniform	no cache	160	13.58
		$\log n$	30	2.517
100	Zipf	no cache	475	35.8
		$\log n$	60.2	4.48
	Uniform	no cache	312	27.3
		$\log n$	42	3.7
150	Zipf	no cache	454.18	33.8
		$\log n$	106.29	7.6
	Uniform	no cache	368.77	32
		$\log n$	—	—

Table 1: Emulab benchmark results

is highly skewed, and is used to model popularity. Some values in this distribution are extremely popular, and most values are not very popular. For our experiment, we used a Zipf θ value of 0.95.

We also ran our experiment using either no cache, or an LRU cache of size $\log n$, where n is the number of nodes in the attribute hub. This size was chosen as a tradeoff between cache size and staleness of entries. For each publication, the time taken between serialization and matching was recorded, as well as the number of hops the publication travelled to reach its rendezvous point.

Correctness was evaluated by having each application log the publications that were generated, and by having the routing layer log the publications that were matched. A Perl script was used to aggregate results, and see if there were any publications which were generated but not matched. In all of our tests, all generated publications were correctly matched in the system, verifying the correctness of our implementation. Performance results are presented in Table 1.

These results clearly show two things:

- **The overhead incurred from routing and serialization is tolerable for "smaller" node counts.** Between 50 and 100 nodes, the Mercury overhead added is either below the 50ms required for real-time FPS games, or slightly above it. This means that, in its current implementation, Mercury could be used for a real-time multiplayer game on a network with low link delays, and add very little noticeable lag to the user playing the game. Since current servers for FPS games top off at around 64 players [19], our implementation demonstrates that a distributed architecture can be used to push the number of simultaneous players to around double what the current limit is, by incurring a slight performance hit.

- **Caching greatly improves routing efficiency.** As mentioned in Section 3.2, caching is an optimization to the routing protocol designed to reduce the delay in matching publications, and reduce the average number of hops each packet takes before it reaches its rendezvous point. These numbers clearly show that caching achieves the desired effect, by reducing the average delay in matching a publication by approximately 82% (for 50 nodes) and 87% (for 100 nodes), and reducing the average number of hops taken by a publication by 82% (for 50 nodes) and 86% (for 100 nodes).

5 The Framework

The framework was designed to serve as an intermediary between an application which wants to use a publish-subscribe system, and the actual implementation of that publish-subscribe system. The framework is responsible for keeping track of all of the world objects in the current view or context, acting as a local database of objects for the application. Functions are also present in the framework which provide more direct access to the routing layer, in the event that an application needs this kind of control.

5.1 Framework Class Structure

The framework is comprised of several different classes, and they are:

- `RNObject`: `RNObject` stands for *replicated, named* object, and this class is the master base class for all of the objects distributed in the network. Each object has a globally unique identifier (GUID) in order to ensure that replicated copies of the object all refer to the same world object. Globally unique identifiers are a tuple of {host ip, host port, local ID}, which guarantees that objects created on two separate hosts will have unique identifications, as long as hosts don't assign duplicate local IDs to objects. Objects also have a replica flag, signifying whether or not a node owns an object. Replicated objects are not allowed to be published — only the owner of an object is allowed to publish it. In the context of a distributed game, this means that only the owner of a player object can publish that player object. This makes sense in our context, since a player would not want other players controlling him. Currently, an object can only have one owner, but future revisions of MERCURY may allow for multiple owners of an object.

  Each object also has a type associated with it, which allows the object to be typecast to the appropriate C++ subclass. Finally, each object has a "dirty" flag signifying when one of the object's data members has changed, and the object is ready for re-publication.

  The `RNObject` class also provides a unified interface for serializing and deserializing objects, enforcing the policy that all objects must be able to be serialized for delivery over the network, and deserialized for recovery from the network. The actual serialization code is left up to the object, and can be automatically generated using a C++ preprocessor by examining an object's C++ header files. Currently, we have written a Python script to implement this preprocessing step, based on a formal specification of a classes data members.

- `ObjectRegistry`: The `ObjectRegistry` is a storage container for `RNObject`s. Any objects which are received from a subscription are stored in this class, and any objects which the application publishes on the network are published through this class. For example, in a simple game, a player object which is published on the network would be stored in the object registry. Currently, this class is implemented as a hash table, using an object's GUID as the hash key. This class also supports iteration, so new objects with unknown GUIDs can be found.

- `Controller`: The `Controller` is the heart of the framework. It provides the means for communication between the application and the routing layer, and it is responsible for maintaining the `ObjectRegistry` of world objects by publishing objects when they are dirty, and updating objects when new publications arrive from the network. Objects become dirty when they are modified by the application. The controller is also responsible for passing subscriptions from the application to the routing layer.

  One special implementation detail about the controller is that we enforced a policy of having all nodes first join the attribute hub before sending publications, when running our experiments in Section 4.8. This policy is strictly an implementation detail, and is not specific at all to the MERCURY specification. This policy was needed in order to guarantee accurate benchmarking results; if nodes were allowed to publish immediately after joining the attribute hub, then publications would be routed before all of the nodes joined the hub. This would introduce incorrect data into our results, as the hop counts would be lower for the packets which were sent before all nodes joined the attribute hub. Thus, we enforce that no publications are routed until after all nodes have joined the attribute hub.

  The way we enforced this policy was by making the controller drop all publications from the application until the SIGUSR1 signal was received. This signal can be sent to an application by using the "kill" command, and a Perl script was written to iterate through all running MERCURY nodes sending them this signal, after all nodes had joined the attribute hub. This way, publications were only routed until after all nodes joined the attribute hub.

- `Value`: In order to handle all of the primitive types which MERCURY supports, while at the same time not having to write separate cases for each type, the `Value` class was created. This class simply serves as a wrapper for the primitive types that MERCURY supports (`char`, `int`,

`float`, `double`, `string`). An example of an instance of this class is: {50, int}.

This class also implements comparison operators, such as less than, greater than, equals, etc. used for comparing two different instances of the `Value` class. Finally, in order to correctly split an attribute space in half, as is required when one node joins another, this class implements a static function which calculates the "distance" between two values. For all types except string, the distance between A and B is computed by $\text{abs}(B - A)$. For two strings, str_1 and str_2, a different algorithm is used:

1. compute the sum totals s_1 and s_2 of all of the letters in the string, by converting each character to an integer (ASCII decimal value, $a = 97, b = 98, \ldots$, and summing the integers
2. compute values v_1 and v_2 by the following expression, where $j \in \{1, 2\}$:

$$v_j = \sum_{i=0}^{len(str_j)-1} \frac{2^{(len(str_j)-i)} * str_j[i]}{s_j}$$

   where $str_j[i]$ is the integer value of the i^{th} character in string str_j
3. return $\text{abs}(v_2 - v_1)$

This algorithm was created in order to judge the distance of two strings based on the differences in distances in the individual letters, weighted by where the letters occur in the string. For example, the distance between the strings *abc* and *bcd* is only 2.06×10^{-4}. The distance between *mercury* and *caduceus*, which intuitively seem farther apart than the previous example, is 25.14.

- `TypedAttrValue`: The `TypedAttrValue` is an extension of the `Value` class. Recall from Section 3.1 that each attribute in the system has a name, a type, and a value. This class, therefore, is the representation of attributes. This class contains a string for the name of the attribute, and a `Value`, which contains both the type of the attribute, and the actual value of the attribute. An example of an instance of this class is: $\{\text{x}_\text{c}\text{oordinate}, [50, int]\}$, where the square brackets around "50, int" mean that that data is an instance of the `Value` class.

- `TypedAttrConstraint`: This class represents a single constraint in a subscription. Each subscription is a conjunction of typed attribute constraints, each of which contain a type, attribute name, operator and value. Thus, this class contains a string for the name of the attribute, and the operator and `Value` used for making the constraint. An example of an instance of this class is: $\{\text{x}_\text{c}\text{oordinate}, <, [50, int]\}$, where the square brackets around "50, int" mean that that data is an instance of the `Value` class.

- `Event`: An `Event` is a publication, and is composed of a list of `TypedAttrValue`s. This class is used to store the publication data inside of the `PubsubData` class discussed in Section 4.2.

- `Interest`: An `Interest` is a subscription, and is composed of a list of `TypedAttrConstraints`s, as well as a flag signifying whether or not the subscription is really an "unsubscription". Recall that if a subscription's unsubscribe flag is set when matched, any subscriptions matching that subscription are removed from a node's subscription list. This class also keeps track of who created the subscription, so publications which match the subscription can be delivered to the subscriber. This class is primarily used to store the subscription data inside of the `PubsubData` class discussed in Section 4.2. The `Interest` class also has functions to determine whether or not a subscription should be routed to the left, right, or stored at the current node, based on what a node's minimum and maximum node range values are. This makes the process of determining how a subscription should be routed much easier for the router.

- `Handler`: In Section 4.5, a slight departure from traditional MERCURY routing was discussed, whereby an application gained a greater degree of control over the specific routing of a publication (or subscription). The `Handler` class is the class responsible for passing publications and subscriptions up to the application, after they have been matched at the routing layer. This class has two functions, one which is called when a publication is matched, and one when a subscription is matched. Both of these functions return a boolean value, signifying whether the publication or subscription should be routed normally, or whether the application took care of it.

This class is subsclassed and implemented in the application layer.

- `XMLManager`: In order to minimize the complexity of using the Xerces C++ parser, the `XMLManager` class was created to provide a simple front-end to XML serialization. The XML manager contains functions to convert C-style strings into XML-style strings (`XMLCh *`), as well as functions used for parsing XML data.

The way an application interfaces with the controller is very simple. First, the application creates an instance of the `Controller` class. The controller constructor then creates an instance of the `MercuryRouter` class. When the application is ready for the routing layer to perform the bootstrap process, the application tells the controller to start, and the controller tells the router to bootstrap. After the bootstrap process has completed, control returns to the application. However, publications the application generates at this point will not be routed until the application process receives the signal SIGUSR1. The reasons for this are discussed in Section 5.1.

Once the SIGUSR1 is received, routing begins. When the application wants to distribute one of its objects onto the network, the application registers that object with the controller. Every so often (as often as the application wants), the application can tell the controller to synchronize its state with the network. Any non-replica objects which have been locally modified will be serialized and published. Any publications received from the network will be deserialized, and the object registry will be updated appropriately. New objects will be added to the object registry, and existing objects will be updated with the new published values.

This is how the entire system operates, and this walkthrough shows the interactions between the three layers of the system. The framework was designed from the beginning to provide a very simple interface to the application, freeing the application from the specifics of dealing with a publish subscribe system. The system evolved based on unforeseen needs, but the end result is a very workable framework which a wide variety of applications can use to interface with a publish-subscribe routing system.

Now that the fundamental operations of the system have been discussed, and the connections between application, framework and routing have been exposed, the next section details the implementation of our real-time multiplayer game, Caduceus.

6 Caduceus

Caduceus is the name of our real-time multiplayer game. It is based off of the popular KAsteroids [9] game, and it is built on top of the Qt/X11 Free library [18]. The Qt library provides a GUI environment for the game, as well as a sprite toolkit for the display and animation of game sprites. Using a sprite kit allowed us to solely focus on the conversion of the single player KAsteroids game into the multiplayer game Caduceus, without having to worry about the intricacies of quickly blitting sprites to the screen, or dealing with the collision detection between sprites.

KAsteroids is a game where the player flies around in a space ship, and shoots moving asteroids with a missile to gain points. The player has to be careful enough not to get hit by an asteroid, otherwise a life is lost. When the player runs out of lives, the game is over. There are various powerups which can be obtained in order to help with this venture, such as shields, energy (which is consumed by firing missiles and maneuvering the ship with its thrusters), the ability to shoot more missiles in succession, a teleporter and brakes which stop the ship instantly. After the asteroids on the current level have been cleared, the player advances to the next level, ad infinitum, until the player has lost all of their lives[8].

In order to transform this single player game into a multiplayer one, a few changes were made. First, the notion of "levels" was removed, as the purpose of our multiplayer game is to fly around shooting at your opponents; a cooperation mode where players needed to work together in order to shoot the asteroids would have also made for an interesting game, but is not the focus of this work. Second, all of the asteroids, powerups and shields were removed. This makes the game much simpler to implement, because there are less world object types to handle. Finally, the size of the world map was increased, in order to give space for the many potential players in this game world. This latter change was the hardest to implement, as it involved making many coordinate transformations in order to get the player's ship displayed in the center of the screen, giving the player a view of the world around their ship (that moves with the ship), rather than giving the player a fixed view of some portion of the game world, that does not update when the player moves.

It should be noted that even though Caduceus

[8] As hard as I have tried to find one, I am not entirely sure that there is an end to this game!

is a 2D game, the underlying routing architecture used only creates one attribute hub, for the attribute "x_coordinate". Thus, the world is divided into vertical slices, based on the number of nodes in the attribute hub.

This leaves us with a very nice laboratory to work with, in order to satisfy two main goals:

- To write an application on top of the framework in order to verify that the framework provides an adequate interface to the application layer.
- To learn what cheating behaviors can be executed in a real-time, distributed, multiplayer game.

We begin with a presentation of the main classes in CADUCEUS, and then describe some of the issues brought out during CADUCEUS's implementation.

6.1 Caduceus Class Structure

CADUCEUS is composed of many different kinds of classes. Some classes are subclasses of the Qt `QCanvasSprite` class, and are responsible for displaying different kinds of sprites on screen, such as explosion bits, ships, missiles and ship exhaust. Other classes deal with the GUI and are responsible for reading the keyboard state, and for displaying the main game window. The rest of the classes deal more directly with the mechanics of the game, and they are:

- `KAsteroidsView`: This is the main class of CADUCEUS, and it is the one which instantiates the `Controller` class. This class is responsible for displaying world objects in the current game view, and for generating publications and subscriptions based on user input.
- `KSpawnRequest`: Before a player can participate in the game, they need to send out a spawn request to a node in the attribute hub. When a node receives a spawn request, a `KShip` object is created and returned back to the requesting node. This spawn request contains the nickname with which the player would like to be identified, as well as the local ID that should be used for the returned ship. The reason for specifying a local ID in this class = is described in Section 6.2, and deals with being able to identify which ship was returned from the spawn request. Since this class must be serialized and sent over the network, it inherits from `RNObject`.
- `KShip`: As has been the trend, each layer in our system has one or two classes fundamental to the structure of that layer. In this layer, the `KShip` class is of this type. Each player in the game has their own `KShip` object, since without a ship, a player wouldn't be able to play the game! The `KShip` class keeps track of a ship's name, its x and y location, its velocity and angle, whether or not it fired or died (and the time of the last firing), whether or not it is thrusting, the number of missiles it has in play, and its power level. All of this information is used to display the ship on screen, move the ship on screen with some velocity, draw missiles if the ship fired, explode the ship if it died, determine whether or not a ship is allowed to fire based on the time of the last firing, etc. This class must be serialized and sent over the network, so it inherits from `RNObject`.
- `KMissile`: When a ship fires a missile, an instance of this class is created. The `KMissile` class is responsible for keeping track of which ship fired the missile, based on GUID, as well as the location and velocity of the missile, the age of the missile, and whether or not the missile is still active or has expired. Since this class must also be serialized and sent over the network, as with the `KShip` class, it inherits from `RNObject`.
- `KKeyboardState`: A player's ship moves based on the keyboard keys that are pressed. This class captures the state of the keyboard, and allows it to be serialized for transport over the network. Relevant keys are: ship turning left and right, thrusting, firing, and committing suicide. While the practicality of having a suicide key which instantly explodes one's ship might be questionable, this feature was implemented as a means of testing ship explosions without requiring more than one ship in the game.
- `KGameWorld`: As is discussed in Section 6.2, the `KGameWorld` class is responsible for running a slice of the game world. This class has its own `ObjectRegistry` as well as game window (called `KGameWindow`, which simply contains a canvas for drawing, to perform collision detection). This class is a subclass of the `Handler` class, and implements the required methods for intercepting publications and subscriptions when they are matched.

6.2 Application Details

The process of playing CADUCEUS from the application layer perspective is presented in Figure 7.

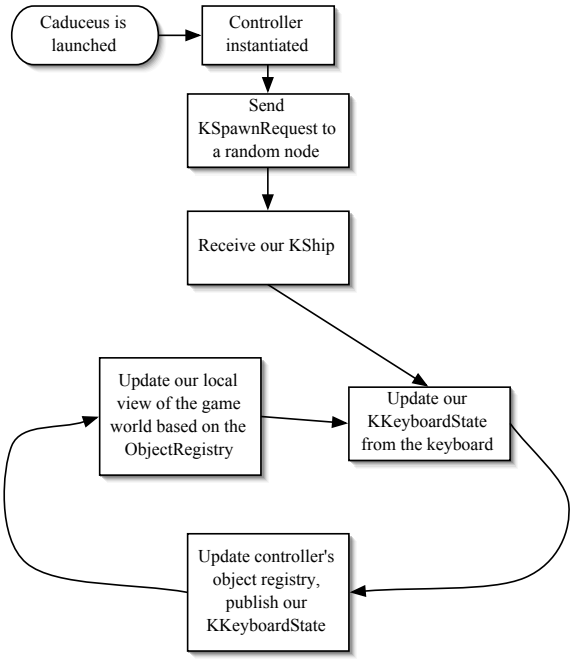

Figure 7: Caduceus flow diagram

When the application first starts up, the controller is instantiated. The user is then presented with a textual message stating that they must press the "J" key to join the game. Upon doing so (and after sending the SIGUSR1 signal to the application), the controller causes the routing layer to bootstrap. When bootstrap is completed, the application creates and sends a `KSpawnRequest` to a random node in the attribute hub. This is accomplished by assigning a random value to the "x_coordinate" attribute, and sending the publication on its way.

When the rendezvous point creates our `KShip` object, it also creates and injects a subscription into the network for the ship's visible portion of the world. Thus, the ship object will be returned to its owner, and because the owner was thoughtful enough to tell the rendezvous point what the *local_oid* field of the GUID should be, the owner knows which ship is his, out of all the other ship objects he may have.

Now that our player has his ship, the time comes for the player to control his ship. At every iteration of the main loop, the state of the keyboard is read in and stored in a `KKeyboardState` object. This object is then published (as long as there is at least one key pressed), and sent to the appropriate rendezvous point (i.e. game server) by adding in the current coordinates of the player's ship object.

This now brings us to the events which occur at the rendezvous point when a keyboard state is read. In Section 3, we presented two different ways for thinking about the role of a publish-subscribe system. To recap, a publish-subscribe system can be seen as a distributed database, with each node in charge of objects that have attribute values falling in a certain range. The second view is to think of a publish-subscribe system as a method of content delivery. However, given that we are creating a game on top of a publish-subscribe system, there is a third way in which this system can be viewed. Each rendezvous point in the attribute hub can be seen as a miniature game server, responsible for only a certain portion of the game world. This view makes it clear what the role of the rendezvous point is in determining the correct course of events in the game, and this is exactly how consistency in game state is achieved.

Maintaining Game Consistency

Consistency is defined as having multiple views of the game world agree with one another. For example, if two players participate in a game world, and both players agree upon the locations of themselves and the other, then that game world is said to be *consistent*. If, however, one or both players do not agree on their respective locations, then that game world is said to be *inconsistent*. From the perspective of a multiplayer game, it is highly important to present a consistent view of the game world to each player, as their decisions in the game world are directly dependent on their perceptions of the game world. Games are sometimes said to go "out-of-sync" when the consistency of the game world has been lost.

In an early version of CADUCEUS, the entire process depicted in Figure 7 was non-existent. Rather, a player simply published their own ship object each time it became dirty. The early versions of the cheating detection algorithm presented in Section 7 were based upon the fact that each player would be publishing their own ship object.

However, allowing each player to dictate the location of their ship, along with the state of their ship, causes many difficulties for maintaining consistent game state. For example, because of the delays inherent in any networked application, one player may see

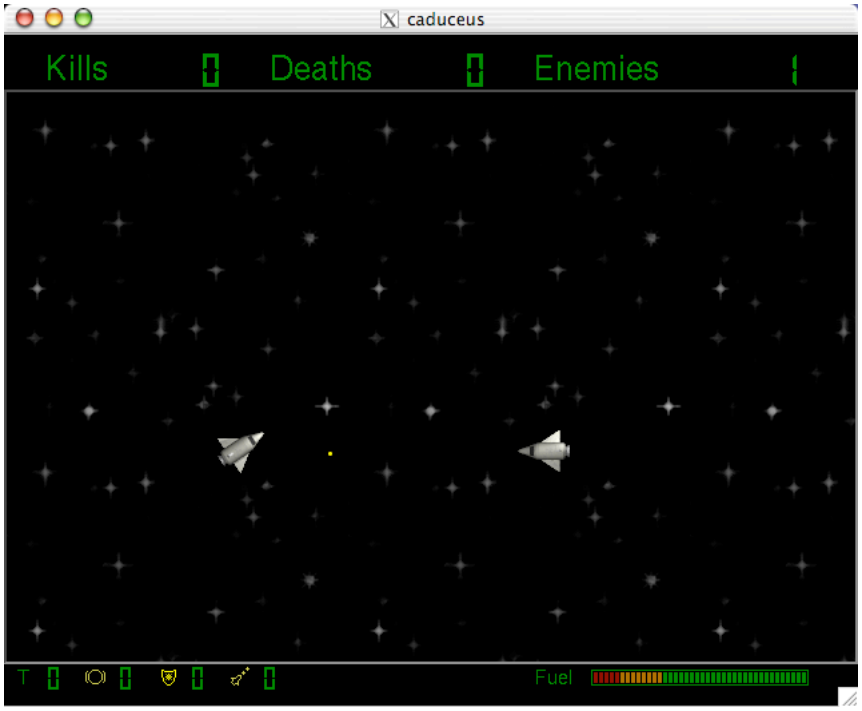

Figure 8: Caduceus action shot!

that ship A shot a missile and hit ship B before a missile shot from ship B was able to hit ship A, whereas another player may see the exact opposite. This is a problem because A and B's actions are dependent on what they perceive in the game world. Both A and B expect that their actions will have a predictable consequence, and a lack of game consistency prevents A and B's actions from being predictable, since there are two inconsistent views of the game world.

In light of this problem, maintaining game consistency *in our implementation* quickly became the top priority. In order to maintain game consistency, we adopted the view of having each rendezvous point act like a miniature game server, and give authoritative control of all world objects to the rendezvous point. This led to the creation of the `KGameWorld` class, responsible for keeping track of the actions occurring in the section of the world controlled by the rendezvous point. This also led to the decision to have each player send keystrokes to the rendezvous points, rather than entire ship objects. As it turns out, this is also how Quake works; players send keystrokes to the server, and it is the server's job to run the game world, and move all of the objects around.

Thus, due to the different types of communication among nodes in the game environment, either sending keystrokes or the positions of ships, we studied cheating behaviors and cheating detection in both of these environments. Section 7 discusses the cheating behaviors we identified in these environments, and presents algorithms for detecting when these behaviors are occurring.

With the rationale for sending keystrokes established, the process for updating a ship object at the rendezvous point given a keyboard state is as follows. Once the keyboard state for a particular ship has been read in, that ship object's location, speed and angle are simply updated at the rendezvous point (in the `KGameWorld`, at the application layer). The rendezvous point is responsible for applying physics of movement to the ship, as well as creating any new `KMissile` objects if the ship has fired. Also, if a ship has committed suicide, the rendezvous point will update the appropriate flag in the ship object.

After these updates, the rendezvous point will update its own sprite world, moving all ships and missiles based on their velocities, and then determining if any ships were hit by missiles using the collision detection functions present in the Qt canvas class. If this event has occurred, the affected ship objects are marked as dead, and all changed objects are published directly to their subscribers.

The rendezvous point also checks to see if any objects moved out of its range, and if so, it removes them from its object registry. When the rendezvous point publishes an object out of its range, it will simply be routed to the correct rendezvous point for that object, and that object will be added to the new rendezvous point's object registry. This is an implicit handover scheme which can be easily exploited, such as when a node generates its own `KShip`s directly (without making a spawn request), and handing them over to his MERCURY predecessor or successor. This situation is discussed in more detail in Section 7.2.

The game view presented to a player is of a fixed size, yet the actual game world they participate in is much larger. Subscriptions are used to both filter out information not relevant to a ship's current view, and filter in information which is relevant. Thus, when a player's ship object moves in the world, the player's subscription to the game world must also change along with the movement. The `KGameWorld` keeps track of the world subscription for each ship, and sends an "unsubscription" message (for the old view) and subscription message (for the new view) each time a player's ship object moves. The unsubscription message is just a subscription with it's unsubscribe flag set, signifying that the subscription is to be deleted from the rendezvous point.

Finally, after the new ship is published, and the ship's new subscription is sent, the cycle repeats itself; the player can send more keystrokes, and the rendezvous point will update the game world accordingly.

Since there is one central authority responsible for running a small subsection of a world, consistency is achieved. However, this central authority has a player on the same node, with a vested interest in the course of game actions. Thus, there are many ways in which this player can act abuse his authority and act in a dishonest fashion, bending the rules of the game for his own benefit. These cheating behaviors, and methods for detecting when they are occurring, are detailed in Section 7.

6.3 The Pitfalls of Relying on Other People's Software

An example screenshot of CADUCEUS is shown in Figure 8. This screenshot was taken from an earlier version of CADUCEUS which did not address the consistency issue. The current implementation of CADUCEUS addresses these issues, but suffers from a few bugs and currently does not work as well as the previous version did. The source of these bugs are either in the Xerces library itself[9], or in how we are using the library. Thus, as we are dependent on other people's software, we are currently looking into how to work around these bugs.

7 Cheating and Cheating Detection

With the complete gaming platform established, we consider two situations in which cheating behaviors can occur; the first is *player-centric*, in which the game world does not possess a globally consistent state, and the second is *server-centric*, in which the game world does possess a globally consistent state. In the server-centric case, the "server" refers to the rendezvous point to which publications are routed, to establish a view of a subsection of the entire game world. While the current implementation of CADUCEUS uses a consistency preserving structure, we feel it is beneficial to examine both approaches to cheating detection.

When developing any algorithm for detecting malicious behavior, it is important to know who the algorithm is being designed for, and where the algorithm will be run. In any multiplayer game, where there are any number of human players participating in the virtual game world, the parties most affected by cheating are individual players (as opposed to, say, the operator of the game server who is not playing in the game). Thus, because each player has a vested interest in making sure that other players are not cheating in order to gain advantage over them, we placed cheating detection in the hands of the players.

7.1 Player-centric cheating: Cheating in the Absence of Game Consistency

In this situation, the publication of a player's ship[10] is controlled by the player. Subscriptions are also controlled by the player, since they must know what events occur in their game view.

[9] During the course of implementation, we have found bugs in Xerces, and later discovered that newer versions of the library fixed these bugs. It may be the case that the bugs we are experiencing here are currently being fixed, but as we are not entirely sure what the specific problem is, we have been unable to determine what the fix is.

[10] In CADUCEUS, each player is represented as a ship, but this analysis is relevant to any player object having an x and y (and z) location in a game world.

Under the assumption that a player has full control of the publications and subscriptions that are sent out, this situation allows for some very interesting behaviors:

- **A player can forge publications.** There are two ways in which this behavior can be disruptive. The first is when a player forges publications of their ship which violate the constraining physics of the game world. This allows a player to travel at a speed faster than the maximum allowed, or even teleport (move instantaneously) to any arbitrary location in the game world. A clever cheating player could even carefully craft a violation of the game's physics in such a way as to make it indistinguishable from a high-latency situation in which game updates are sporadic and non-constant.

  Forged publications would also allow a player to "clone" themselves, potentially causing a replication of their ship object across different rendezvous points. Since the rendezvous points do not have any control over the actual game world in this situation, a cheating player with control over his publications has a lot of freedom to gain advantages over other players, such as: appearing and disappearing at whim, or presenting clones of their ship in front of every opponent in the game, firing missiles from all cloned ships simultaneously. This type of cheating, while certainly possible, has not yet been seen in multiplayer games, and as such may be novel to our particular game architecture.

- **Faulty subscriptions.** A player can make and send subscriptions for parts of the game world that are not visible in their current game view. Since there is no authority validating the subscriptions that are sent out, a player can simply subscribe to the entire game world, and receive the information needed to launch a successful attack against the entire game world. In other multiplayer games, this cheat is known as a "wall-hack" or "map-hack", because it allows players to see through walls, or to view the entire world map.

  The specific mechanics of a traditional wall-hack are different in FPS games than in this situation, but the end result is the same — the player is exposed to more information about the game environment than they are privy to.

Detecting Violations in Game World Physics

Detecting when a player is cheating by manipulating the publishing process, such as having clones in multiple locations at once, reduces to a problem of verifying that a particular player is obeying the physics of the game world. Thus, the cloning behavior can be detected because a player cannot be at more than one location simultaneously, since being in two or more places at once is a clear violation of the game world's physics.

We present the following algorithm for detecting when a player C (potential cheater) is violating the physics of the game world:

1. On some trigger for a player C,
2. Compile a list of the recent publication history of C and,
3. Run this list through an artificial neural network (ANN) which has been trained to detect suspicious behavior.
4. If the neural network classifies the publication history as suspicious, increment a counter for player C, and report this change to the player who is running this algorithm.

Call the player who runs this algorithm D (detector). In the first step, it is up to D to determine when to run the algorithm on player C, to determine if C is cheating. There are a few different choices for this trigger, and each choice is able to detect certain types of behaviors, but may miss others. They are:

- **Trigger the algorithm on player C after n seconds have elapsed**: Like polling, this solution is not ideal as it can use enormous amounts of CPU time and consume too much network bandwidth if run too frequently (when n is on the order of milliseconds), or alternatively, cheating behaviors may be missed if n is too large (on the order of seconds).
- **Trigger the algorithm on player C after he kills D**: This solution seems to avoid the errors of the previous one, assuming that the time between D's deaths is large enough. However, this does not prevent a situation in which a player simply violates the physics of the world without killing anyone. For example, a player could instantly teleport into D's view, and not be caught (by the cheating detection algorithm, since D

might just see that something suspicious happened in his game).

- **Trigger the algorithm on player C when he enters D's game view**: This situation allows for detecting a player who has broken the laws of physics when entering another player's view. However, it still does not prevent a player from violating the game's physics when there are no other players in the cheating player's view. This scheme could also consume enormous amounts of CPU time and network bandwidth if there are many players entering D's view in a short amount of time.

The next step in the algorithm is to compile a list of the publication history of player P. In order to create a full publication history, each node in the attribute hub[11] would have to be queried for its publication history of player P. This also implies that each node in the attribute hub needs to store some amount of publication history for each player. Since our cheating detection is only concerned with the violations of game physics, this history can be of the form {timestamp, x location, y location, ship angle}, and storage of this information should not be a major factor. It should be noted that compiling a full publication list (from every node in the attribute hub) is the only way to guarantee that a player hasn't published their ship object in multiple locations simultaneously. However, requesting and receiving all of this information is a $\mathcal{O}(n)$ operation, where n is the number of nodes in the attribute hub.

Assuming that the first two steps in this algorithm can be performed in a reasonable amount of time, the next step is to determine whether or not a given publication sequence is valid. A neural network was selected for this step because neural networks are able to capture patterns in large amounts of data without having any knowledge of the domain the data came from.

In order to feed the data into the neural network, there is one preprocessing step which must be performed. Each *actual data point* is a tuple of {timestamp, x location, y location, ship angle}. Since we want to learn what kinds of movement sequences are valid in a given amount of time, the actual data point is converted to a *delta data point*, of the form {change in time, change in x distance, change in y distance, change in ship angle}. After this preprocessing step, we run the data through the trained neural network, which reports back a classification as to whether or not the data point is valid, based on the rules the network learned during training. More information on the actual training of a neural net can be found in Section 7.3.

The final step in this algorithm is to present the results to the player who ran this algorithm. Because neural networks have some measure of statistical uncertainty, we expect the misclassification percentage to be strictly positive. Thus, a non-cheating player may be classified as cheating because of missing publication data. To avoid this situation, a player can be presented information with the number of times a player has been classified as cheating. This information can be presented to the user in a separate window, using a graphical histogram format. Players who are cheating will then stand out among players who are not cheating in this way, since the histogram bar will be significantly higher for cheating players.

This strategy provides results in a global fashion as well. If a significant portion of users all notice that a certain player has been classified as cheating, this provides strong justification for those players to take action against that player, such as kicking him out of the game.

Preventing Faulty Subscriptions

In any game environment, a player should only be allowed to see what is immediately visible to him. In our environment, a player's field of vision is a fixed sized rectangular area centered on the player (or flush with the edges of the game world in the case where a player is at the edge of the world). When players are allowed to create their own subscriptions to the world, they can arbitrarily subscribe to any section of the world they choose, or multiple sections of the world, or even the entire world itself.

To prevent against this, there needs to be an authority who checks the subscriptions which are sent out, accepting them only if they are centered on the player, and are a fixed size. This can be accomplished by borrowing an idea from our implementation of Caduceus, by giving a small amount of authoritative power to the rendezvous points.

When a subscription from player P is received at a rendezvous point R, R can check to see if P has published himself as being at R (since R stores the recent publication history of P in accordance with the previous cheating detection algorithm). If R does not have

[11] We assume only a single attribute in this example, "x_coordinate".

P in its publication history, then P is trying to subscribe to a part of the world which he is not privy to, and the subscription should be rejected. Otherwise, R can verify that the received subscription meets the requirements for validity, and either accept or reject the subscription. R can also generate unsubscription messages when a new subscription is received, to ensure that any one player only has one subscription in the network at any given time.

This concludes the discussion of cheating detection in a world where players are responsible for sending updates regarding their status and location. While algorithms for dealing with the two primary cheating behaviors in this environment have been created, it should be noted that cheating detection in this environment is not an easy venture, and if it is to be done in a reasonable manner (with regards to finite computing resources), it appears that in practice the results can only be described as a "best-effort" solution that may reduce, but may not eliminate all cheating behaviors.

7.2 Server-centric cheating: Cheating in the Presence of Game Consistency

In this situation, a player is not responsible for sending updates regarding his status and location. Rather, there is an authority that controls the state of the world, in order to provide a consistent environment for all players. This authority is the rendezvous point, the node responsible for matching publications with subscriptions for a certain portion of the attribute space.

Each rendezvous point is just a node in an attribute hub. Going back to Figure 5, node D is the rendezvous point for all publications with an "x_coordinate" in the range of $[8.9, 10.0]$. Thus, node D is responsible for keeping track of all events which occur in this space. Let this space be known as a *game slice*, the portion of a world controlled by a rendezvous point.

To control his actions — in essence, to play the game — a player sends the state of their keyboard to the rendezvous point, and the rendezvous point performs the actions dictated by the keyboard state. For example, when the rendezvous point notices that a player has pushed the "up" key, the rendezvous point moves the ship forward, and republishes the ship object.

This leads to a very interesting scenario in terms of what behaviors are possible by malicious players. No longer can a player directly publish their ship object as in the previous scenario[12] Players are also not responsible anymore for the creation of subscriptions, as the rendezvous point will do this automatically as part of its world updating process. Therefore, the player has **no powers whatsoever** in altering the game state of a rendezvous point, **unless they alter the game state of their own rendezvous point**.

Recall from Figure 1 that each node in an attribute hub is comprised of three layers — routing, framework and application. Sitting on top of each rendezvous point is an instance of the game, in which the player participates. Thus, a player who compromises their instance of Caduceus can gain **full, complete control** over the entire workings of their game slice.

A malicious player could conceivably alter many things within their game slice. Examples of what a malicious player could do are:

1. Modify the movements of other ships, and/or prevent them from entering other game slices.

2. Violate the game world's physics, as discussed in the previous section.

3. Kill everything in the game slice, taking full credit for the kills.

These behaviors can still be detected, albeit not prevented, using the same techniques presented in the previous section.

The first and third types of cheating are easily detected by the player himself. When a player's ship ceases to respond to the keys he is pressing, it is clear that something non-kosher is occurring at the rendezvous point. Likewise, when a player notices that they have been killed without any apparent cause, it is a likely deduction that the rendezvous point has been tampered with.

This leaves the second type of cheating behavior, which is a violation of the game world's physics. There is a distinction though, between what is meant by "violation" in this section, and "violation" in the previous section. In the previous section, players were

[12]This is not entirely true in the current implementation. One weakness that could be exploited related to how the handover of objects is performed; no checks are made to ensure that the publication came from a predecessor or successor node. Thus, a player could directly publish a `KShip` object to a specific rendezvous point, and it would be blindly accepted. Future work includes creating a stronger object handover method to prevent this type of behavior from occurring.

able to arbitrarily publish their ship in multiple locations at once. This behavior cannot happen in this situation, as players do not publish their ship objects. Rather, to simulate the cloning behavior in this situation, a player can send multiple spawn requests to different rendezvous points. However, this behavior amounts to simply running multiple instances of the game simultaneously, and is not a type of behavior we classify as cheating.

In contrast, a player can in fact teleport their ship objects into and within their own game slice, which is representative of the type of cheating previously discussed. The algorithm presented in the previous section can be modified slightly to track players inside the current game view, and verify that the game's physics are being adhered to. This new algorithm is:

1. When player P moves,
2. Take the difference in times, x position, y position and angle,
3. Run this instance through the neural network,
4. If the neural network classifies the publication history as suspicious, increment a counter for player P, and report this change to the player running this algorithm.

This algorithm is a lot simpler than the previous one, because checking for multiple publications of a single ship object throughout an entire attribute hub is unnecessary. The next section details an experiment performed to evaluate just how well a neural network can learn the physical laws of a game world, and detect movements contrary to the physical movements permitted by those laws.

The greatest drawback in using the MERCURY system for multiplayer games is that it leaves some authoritative control of the game world in the hands of those who have a vested interest in abusing that power. The entire game world is run on machines belonging to the players participating in the game world; thus, players who wish to gain an upper hand on their opponents may be able to abuse their powers, in their own little corner of the world. Several algorithms designed to detect when these behaviors are occurring have been presented, but these algorithms do little else to actually prevent such behavior from occurring.

To fulfill the goal of designing a large-scale distributed game system not controlled by one central authority, these authoritative powers must fall into the hands of the players using this system. But, this leaves the system vulnerable to the whims of malicious players who would abuse these powers. Thus, the algorithms presented in the previous sections seek to find a middle ground to this dilemma, by allowing honest players to detect when they are being abused by malicious players, and giving them the information they need to decide if they should take corrective action or not.

7.3 Neural Network Evaluation

To determine how well a neural network can detect violations of our game's physics, the following experiment was performed. Movement traces were obtained from a virgin copy of KAsteroids, where each trace was a list of {frame number, x position, y position, ship angle}[13] As mentioned earlier, these data points are the *actual data points*. Two categories of traces were created, valid and invalid, where the invalid traces contain violations of a game world's physics.

Valid traces were created from the actual data points in the following fashion:

- Random lines from the movement traces were removed, to create uneven movement and losses similar to what would be seen in a game with high latency.

Invalid traces were created from the actual data points in the following fashion:

- Lines were permuted randomly to create a teleportation effect.
- Data points in the traces (x, y, angle) were changed by small random amounts to give the effect of an increase in ship speed which would cause the ship to move faster than the actual limit imposed by the game's physics.

All traces were then postprocessed to create a list of the form {change in time, change in x position, change in y position, change in angle}. As mentioned earlier, these data points are the *delta data points*, and are the data points which are used when training and testing the neural network. Data was partitioned into three sets: one for training, one for validation

[13] KAsteroids updates the game view in a loop. At each iteration of this loop, the ship, rock, powerup and missile sprites are moved. Each iteration of this loop is known as a *frame*, and information about the ship's position is captured to a file for each frame.

Correct classifications	9890	80.9%
Incorrect classifications	2330	19.1%
Total:	12220	100%

Table 3: Classification results of the neural network on the training data.

during training, and one for testing. The validation set was used to determine when training the network should cease, based on the network's performance on the validation set. A breakdown of the sizes of each data set is given in Table 2. More information about neural network training, the problem of overfitting, and how the use of a validation set prevents overfitting can be found in [1, 21].

In order to train and test a neural network, we used the MATLAB® neural net toolbox [11]. We used a feedforward, backpropogation network with 3 layers, one for input, one hidden layer, and one for output. A sample diagram of the network we used is shown in Figure 9. The input layer contains four inputs, one for each piece of information in a delta data point. The hidden layer has 20 units. This number was found experimentally, based on training and testing networks with different hidden layer sizes. Choosing the correct number of hidden units for a neural network is still an open problem, and is discussed in further detail in [6].

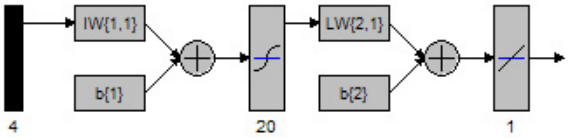

Figure 9: Neural network layout. This network has four inputs, one for each item in a delta data point, 20 hidden nodes, and one output. The `traingdx` method was used for training; GDX combines gradient descent with an adaptive learning rate and momentum training. This method typically converges more slowly than other training methods, and can help prevent overfitting when using an early stopping mechanism. More information about this method can be found in Chapter 5 of [11]. A tan-sigmoid transfer function was used at the hidden layer, and a pure linear transfer function was used at the output node.

A graph of the neural net training performance can be found in Figure 10, and the results of the neural network's performance on the testing data can be found in Table 3. Since the network's output unit can produce any value, a binary threshold was applied to the ouput value. Negative outputs were mapped to 0, and positive outputs were mapped to 1. This has the same effect of using a tan-sigmoid transfer function in the output layer, so our choice of transfer function does not affect the results.

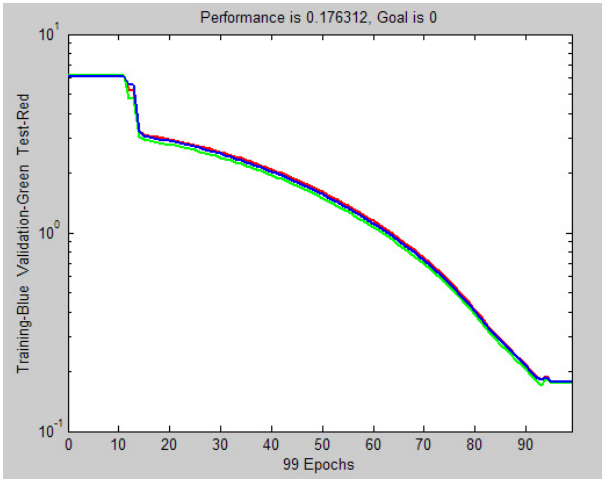

Figure 10: Neural network training performance. The blue line (middle) represents the mean squared error (MSE) of the learned weights on the training data; the green line (bottom) represents the MSE on the validation data, and the red line (top) represents the MSE on the test data. MATLAB® allows the testing data to be used during training as a method of estimating generalizability, and this was found to produce networks which performed better at classifying the testing data.

In classifying the testing data, the network correctly classified 80.9% of the testing examples, and mis-classified 19.1% of the examples. Unfortunately, this network does not perform as well as we expected, and implies that there may be several problems with our experiment. Potential hazards are:

- **Cheating behaviors may not have been accurately captured in the invalid data.** Recall that the invalid training examples were created by adding or subtracting small random values from each actual data point. This may not have been enough to produce a significant modification to the game's physics, so a permutation of actual data points was performed as well.

Training data	Valid	9986 data points
	Invalid	10289 data points
	Total:	20275 data points
Validation data	Valid	7185 data points
	Invalid	7809 data points
	Total:	14994 data points
Testing data	Valid	6171 data points
	Invalid	6049 data points
	Total:	12220 data points

Table 2: Data set sizes. Each data point is a tuple of {change in time, change in x position, change in y position, change in angle}. Changes in time are calculated as the difference in the frame numbers between two actual data points.

While these techniques were created to produce movement which would be impossible to produce when constrained to the game's physics, they might still have produced movements which would be valid within the game. Thus, this would create training examples which are incorrectly identified as being invalid, causing the network to classify future similar (and valid) examples as being invalid. Even though neural networks are robust in the presence of noise, a significant number of these "mis-labeled" examples could cause the network to learn a function different from the one we intend it to learn.

- **Incorrect neural network parameters could cause an inefficiency in learning.** It may be the case that the training method, or transfer function, or the specific parameters MATLAB® uses when performing network training were not optimal for the data we presented to the network. Since different network types perform better under different problem domains, further experimentation is needed in order to determine what type of network best fits our needs.

- **The full physics of the system may not have been captured in the delta data points.** Each delta data point represents a change in position over a change in time. However, a ship also has a *velocity* at every time point, implying that there is some amount of inertia which keeps the ship travelling in the same direction, until the player turns the ship in another direction and applies thrust to change the course of the ship. Future experiments could capture the changes in velocity in between time points, and see if this affects neural network classification at all.

In summary, we have trained a neural network to recognize inconsistencies in movement data. Our data was taken from actual gameplay in the physics environment used by KAsteroids and Caduceus. Some of this data was modified in order to produce movements which violate the physics of the game environment, but this modification may still have produced valid movements, adding noise into the data set, and leading to a degradation in the performance of the neural net in classifying new data. Future work is needed to determine whether the network's poor classification performance is due to either poor generation of cheating data, or an inappropriateness of using a neural network to learn movement physics.

8 Future Work

Future work needs to be done in each of the following areas:

- **Leaving**: Nodes need to be able to leave an attribute hub after they have joined, with the other nodes in the system sensing the break in connection, and automatically reconfiguring themselves to maintain the circular topography required by Mercury. This also means that nodes need to be able to reclaim the lost attribute space when a node leaves the system.

- **Multiple attributes**: Multiple attribute routing has been implemented, but work needs to be done to determine how authoritative control

over objects is delegated in a system with multiple attributes. One thought is to give authoritative control to the attribute hub with the lowest value when lexicographically sorted, although this scheme has not yet been tried.

- **Caduceus**: The implementation of CADUCEUS suffers from bugs described at the end of Section 6.2. Ultimately, these bugs should be fixed, giving us a working, playable version of CADUCEUS.

- **Routing layer**: Many multiplayer games have the notion of a team game, where players form some number of teams which work cooperatively against members of the opposing teams. Future work could examine how to use MERCURY to distribute sensitive information to teammates while keeping it hidden from opponents.

- **Object handover**: When a rendezvous point moves an object outside of its bounds, this object is handed over to the rendezvous point's predecessor or successor, depending on the new value of the object's attribute. This handover process is currently insecure, allowing a malicious player to create arbitrary world objects and hand them over to unsuspecting rendezvous points. Future work involves creating a more secure handover method which would prevent this behavior.

- **Cheating data collection**: One of the limitations of the current study is that actual cheating data was not used in training the neural network. It would be more beneficial if movement data could be collected from an actual FPS game. For example, Quake 2 is an open source project [13] and can be modified to give players the ability to violate game physics, by teleporting randomly across the map, and by running faster than normally allowed. This would guarantee that all data which should be in violation of game physics is actually in violation of game physics.

9 Summary and Conclusions

The recent trends in the video game industry suggest that systems will need to be developed in order to handle more concurrent users than current technology is able to accommodate. Presented in this paper is the publish-subscribe routing protocol MERCURY, which can be used to scale real-time multiplayer games past the current limits, while maintaining the property that the game world is not controlled by one central authority.

In order to build a game on top of the MERCURY system, a framework has been created to serve two purposes: to interface between MERCURY and an application, and to store relevant world state, publishing objects when they are modified from the application, and updating objects as they are published from other sources. Using this framework, an implementation of MERCURY and a benchmarking utility were created, and results indicate that the overhead incurred by using MERCURY routing is acceptable for a real-time multiplayer game.

Finally, because of the problems in current multiplayer games regarding cheating, a real-time multiplayer game CADUCEUS was created to study what cheating behaviors were possible in our system, and if there was any way for a player to detect when another player was cheating. Two different approaches to the management of game state are discussed, either by having players publish their own location and state, or by giving authoritative control of the game world to a separate entity. Several different types of cheating behaviors are exposed for each different approach, and the implications for cheating in both of these situations are examined. Algorithms have been presented to allow players to determine when other players are executing each of these cheating behaviors.

Preliminary results show that the neural network designed to classify a player's movement as either valid or invalid is correct in classifying such movement approximately 80% of the time. In its current state, this network is not suitable for deployment in CADUCEUS, as it would perform many misclassifications, confusing the player as to whether or not his opponents are cheating. However, the poor performance of this network may be due to flaws in the method for creating data which violates game physics, and future work is needed to more accurately determine if a neural network is an appropriate mechanism for learning the physics of a game environment.

Due to the design of the system, which gives individual players authoritative control over a game slice, it is impossible to prevent players from abusing that authority. However, certain cheating behaviors are obvious enough to a player that they can choose to take corrective actions. Other cheating behaviors are harder to detect, such as disobeying of the game world's physics, and as such, the classification algo-

rithm presents its results to the user in a form which allows the user, at quick glance, to determine if everyone in the game world is obeying the game physics. In this situation, individual players can be singled out and avoided, or disconnected from the system.

Thus, one of the most important aspects of cheating detection is to make efforts to assist, and then engage a human to solve a human problem.

References

[1] Backpropogation (Neural Network Toolbox). http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/backpr17.sht%ml.

[2] D. Becker. Online gaming's cheating heart. *CNET News.com*, 2002. Availiable at: http://news.com.com/2100-1040-933822.html.

[3] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the first workshop on Network and system support for games*, pages 3–9. ACM Press, 2002.

[4] Emulab.Net. http://www.emulab.net.

[5] Extensible Markup Language (XML). http://www.w3.org/XML/.

[6] Frequently Asked Questions (FAQ) on Neural Networks 2 of 7. http://www.ipd.uka.de/~prechelt/FAQ/nn2.html#A7.

[7] GameSpy. http://www.gamespy.com/.

[8] A. Joinson. *Psychology and the Internet: Intrapersonal, interpersonal, and transpersonal implications*, chapter titled Causes and implications of disinhibited behavior on the Internet, pages 43–60. Academic Press, Inc, 1998.

[9] The KAsteroids Handbook. http://docs.kde.org/en/3.1/kdegames/kasteroids/index.html.

[10] W. Knight. Virtual world grows real economy. *New Scientist*, 2002. Availiable at: http://www.newscientist.com/news/news.jsp?id=ns99991847.

[11] Neural Network Toolbox. http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/nnet.shtml.

[12] PunkBuster Online Countermeasures. http://www.evenbalance.com.

[13] QuakeForge. http://sourceforge.net/projects/quake/.

[14] E. S. Raymond. The Case of the Quake Cheats. *Eric Steven Raymond's Home Page*, 1999. Availiable at: http://www.catb.org/~esr/writings/quake-cheats.html.

[15] E. Rivera. Movie Theater Offers Big Screen Gaming. *ABCNEWS.com*, 2003. Availiable at: http://abcnews.go.com/sections/scitech/TechTV/techtv_bigscreengaming030%207.html.

[16] G. Sandoval. Sony to ban sale of online characters from its popular gaming sites. *CNET News.com*, 2000. Availiable at: http://news.com.com/2100-1017-239052.html.

[17] D. Takahashi. Zona aims to take online gaming to the next level. *Red Herring*, 2002. Availiable at: http://www.redherring.com/insider/2002/0117/724.html.

[18] Trolltech - Creators of Qt - The multi-platform C++ GUI/API. http://www.trolltech.com/.

[19] The Adrenaline Vault: Unreal Tournament 2003. http://www.avault.com/reviews/review_temp.asp?game=ut2003&page=3.

[20] P. Wayner. Do Cheaters Ever Prosper? Just Ask Them. *The New York Times*, 2003. Availiable at: http://www.nytimes.com/2003/03/27/technology/circuits/27chea.html.

[21] What is overfitting and how can I avoid it? http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-3.html.

[22] XBox Live. http://www.xbox.com/live.

[23] Xerces C++ Parser. http://xml.apache.org/xerces-c/index.html.

[24] xml.apache.org. http://xml.apache.org/.

[25] Zona. http://www.zona.net.