

# **The Fleet Architecture**

Richard W.M. Jones <rich@annexia.demon.co.uk>  
Contributions by Justin Short <justin@annexia.demon.co.uk>

3rd November 1998

## **Abstract**

Fleet is a multiplayer online computer game (MPG). However, Fleet is unlike the MPGs which have gone before in several important ways. Firstly, Fleet disposes of the central controlling authority or server found in almost all previous MPGs: in Fleet player computers communicate peer-to-peer with one another. Secondly, Fleet is designed to scale to 1,000,000 players (both human and computer) in a single game, actively, at the same time. Thirdly, Fleet is a free program, protected by the GPL and supplied with open source: despite this, the architecture of Fleet is designed to prevent players from cheating. Fourthly, Fleet can be extended by players (by, for example, adding new types of weapons or new areas to the game) without requiring changes to the common codebase.

This document, described as the “architecture” document, in fact encompasses several design issues. It contains the requirements and design, the process architecture of Fleet and detailed design for all important subsections.

# Contents

<b>I</b>	<b>Overview</b>	<b>4</b>
<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Requirements</b>	<b>6</b>
<b>3</b>	<b>Architecture</b>	<b>7</b>
3.1	Terminology . . . . .	7
3.2	Overview of the Architecture . . . . .	9
3.3	Sphere of influence . . . . .	10
3.3.1	Notation . . . . .	12
3.3.2	Sphere of influence algorithms . . . . .	13
3.3.3	Updating the influences relationship . . . . .	13
3.3.4	Preventing cheating . . . . .	15
3.3.5	Bootstrapping . . . . .	15
3.4	Multicasting to the sphere of influence . . . . .	15
3.4.1	LAN protocol . . . . .	16
3.4.2	Internet protocol . . . . .	18
3.5	Components and assemblies . . . . .	18
3.5.1	The <b>Component</b> class . . . . .	18
3.5.2	Trusted classes . . . . .	19
3.5.3	The class hierarchy . . . . .	19
3.5.4	The <b>Assembly</b> class . . . . .	20
3.5.5	Component lifecycle . . . . .	21
3.6	Housings and collision detection . . . . .	32
3.6.1	Size and shape example . . . . .	32
3.6.2	The <b>Housing</b> interface . . . . .	32
3.6.3	Collision detection, laser hits, explosions . . . . .	34
3.6.4	Refitting an active with a new housing . . . . .	34
3.7	Status messages . . . . .	34
3.7.1	The <b>ACCEL</b> message . . . . .	34
3.7.2	The <b>ROT</b> message . . . . .	34
3.7.3	The <b>PROT</b> message . . . . .	36

3.7.4	The EXPLODE message . . . . .	36
3.7.5	The LASER message . . . . .	36
3.7.6	The FIRE message . . . . .	37
3.7.7	The FORK message . . . . .	37
3.7.8	The JOIN message . . . . .	38
3.7.9	Network latency and timestamps . . . . .	38
3.8	Policing . . . . .	39
3.8.1	Cheating . . . . .	40
3.8.2	Trusted actives . . . . .	40
3.8.3	Policing the sphere of influence . . . . .	41
3.8.4	Policing status messages . . . . .	41
3.8.5	Policing components . . . . .	42
<b>4</b>	<b>Tools</b>	<b>45</b>
4.1	Languages . . . . .	45
4.2	Libraries . . . . .	45
4.3	Network . . . . .	46
<b>II</b>	<b>Detailed Design</b>	<b>47</b>
<b>5</b>	<b>Persistent server</b>	<b>48</b>
5.1	Active threads . . . . .	51
5.1.1	Associated Active object . . . . .	51
5.1.2	Creation . . . . .	52
5.1.3	Running . . . . .	53
5.1.4	Destruction . . . . .	53
5.1.5	Self-destruction . . . . .	53
5.2	Sending and receiving messages . . . . .	53
5.2.1	Sender thread . . . . .	54
5.2.2	Receiver thread . . . . .	55
5.3	Active objects . . . . .	55
5.4	Policing and updating status . . . . .	56
5.4.1	Immediate policing . . . . .	56
5.4.2	Later policing . . . . .	61
5.5	Rocket mechanics . . . . .	62
5.5.1	Time segments . . . . .	62
5.5.2	Rockets . . . . .	62
5.5.3	Gravity . . . . .	63
5.5.4	Linear motion . . . . .	64
5.5.5	Linear motion, constant rotation . . . . .	65
5.5.6	Rotational engines . . . . .	65
5.5.7	Unit vector function $u$ . . . . .	66
5.5.8	The complete equations of motion . . . . .	67

5.5.9	Solutions to the equations of motion . . . . .	67
<b>6</b>	<b>Front end</b>	<b>68</b>
6.1	Front end to server network protocol . . . . .	68
6.1.1	CORBA as the underlying medium . . . . .	68
6.1.2	Connection, naming and access control . . . . .	69
6.1.3	Overview of the protocol . . . . .	71
6.1.4	Login and feature negotiation . . . . .	72
6.1.5	Status updates . . . . .	73
6.1.6	Commands . . . . .	73
<b>7</b>	<b>Staged implementation plan</b>	<b>74</b>
<b>III</b>	<b>Gameplay</b>	<b>76</b>
<b>8</b>	<b>The universe</b>	<b>77</b>
8.1	The Solar System . . . . .	77
<b>9</b>	<b>Man-made landmarks</b>	<b>79</b>
9.1	Drop-in points . . . . .	79
9.2	Space stations . . . . .	79
<b>10</b>	<b>Spaceships</b>	<b>80</b>

**Part I**

**Overview**

# Chapter 1

## Introduction

## Chapter 2

# Requirements

# Chapter 3

## Architecture

### 3.1 Terminology

We use several precisely defined terms.

The **universe** is where the Fleet game takes place. It has a system of three-dimensional coordinates which allow each location to be uniquely defined. It logically contains all players in play and all objects occupying the universe. However, because Fleet is designed to scale to a universe containing millions of objects and hundreds of millions of possible interactions between objects, you cannot point to any place in the Fleet architecture where the entire state of the universe is known at any particular point in time. The state of the universe is distributed amongst the active objects which inhabit the universe.

The concept of a Fleet **game** is synonymous with the universe. In other words several Fleet games can be going on at once, but they all occupy parallel and separate universes. When the player starts up their Fleet client, they choose which universe in which to play by giving the URL of the **root page** of the particular game they wish to play. The URL of the root page uniquely defines each game (and hence universe). The Fleet client downloads this root page and finds all the information it needs from this page and other pages off it to enter the game.

A player can only enter a game at one of the **drop-in points** listed in the root page for that game. A drop-in point is patrolled by a special object called a **drop-in controller**. This special (trusted) object is permitted to introduce new players into the universe. The root page lists drop-in controllers and their associated locations or drop-in points.

A (human) **player** refers to the person playing the game. A player is represented in the Fleet universe as an object containing several attributes, including a private and public key pair used to authenticate the player to other players, the player's name and email address. **Computer players**, as far as the Fleet

game and other players are concerned, are just like human players. Therefore the generic term “player” refers both to humans and computers.

A player controls a collection of one or more **actives** while they are playing Fleet. An active has a very precise definition: it represents a self-contained autonomous object with a particular location, size and shape in the universe. Examples, in Fleet terms, would be a space craft, a missile, a planet or a space station. Note, however, a subtle distinction: space craft are loaded with missiles, but these missiles do not become actives until they are launched. In other words, when a missile is launched, it becomes a separate, autonomous object, with a location separate from the space craft from which it was launched. At this point, the missile comes into being as an active<sup>1</sup>.

Actives communicate with each other. This communication is peer-to-peer, without any central server. Actives communicate with other actives within their **sphere of influence**, a notional sphere which surrounds each active and encloses all other actives in the universe which the active can see and influence. Actives send network messages between each other. These network messages record changes in **status** (ie. location, direction, speed and so on).

We said that before a missile is launched, it is not an active. What is it, then? In fact, a missile before launch is an example of an **assembly**. Assemblies are, as the name suggests, assembled collections of **components**.

Components are “raw material” items such as fuel, energy and explosives; credits (units of Fleet cash); steel and other metals; **housings** which describe what an active looks like; and **converters** such as triggers and laser cannons. Assemblies are objects which allow convenient collections of components to be traded all at once. For example, a missile is an assembly containing fuel, energy, a housing, steel, some explosive, a trigger, an engine for moving forwards, an engine for rotating, some high-tech components and smarts (the bytecode that controls the missile in flight).

Components have properties similar to digital cash:

- New components are **issued** by special players called **issuers**. Certain components are so important in a game of Fleet (particularly credits) that they may only be issued by **trusted players**.
- Each component instance is owned by a player.
- Components can be **traded** between players, but unlike digital cash, trading is not anonymous. Components are usually traded through a **trading post**. Trading posts often also deal with common assemblies, such as missiles.
- Components are **singular**—in other words, if no player cheats, no two players should believe that they own a particular instance of a component.

---

<sup>1</sup>Active creation is a complex procedure and is outlined in more detail later in this document.

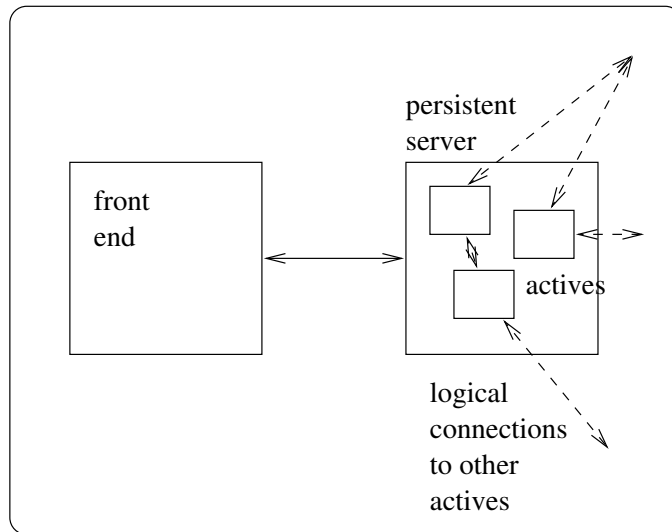


Figure 3.1: Block diagram showing the relationship of the front-end and persistent server components for a single player.

If the situation arises that two (or more) players believe they own the same component, then this has happened because a player has **double-spent** a component—traded it twice with others.

- To ensure that components of the same type can be distinguished, components are issued with a **serial number**.
- At certain times, Fleet requires that components be destroyed. Fleet components are destroyed at special sites (actives) called **destruction posts**.
- Assemblies can be **disassembled**.
- Components contain an **expiry date**. Before this date, they must be replaced (for free) by the issuer.

## 3.2 Overview of the Architecture

As shown in figure 3.1, a Fleet client is normally split into two actual processes—the **front-end** and the **persistent server** back-end<sup>2</sup>. The two halves communicate using a well-defined protocol. The persistent server process runs continuously and manages the set of actives which that player controls—ie. normally

<sup>2</sup>In the case of a computer player, the front-end process is not present.

their space craft and any missiles or other autonomous scouts which they have fired. Each active within the persistent server is logically separate. In other words, it has a separate sphere of influence and logically manages its own connections with other actives (both local and remote). However, the persistent server can optimize this somewhat. In particular, it is optimized for the case where the actives it contains are close together and so mostly exchange network messages with each other. Also, actives can detect that other actives are running locally, and are therefore trusted not to cheat (and so require no security checks).

There is one persistent server per player. In other words, all the actives present in a particular, server belong to the same player, and so all can trust each other.

The main reason for this unusual split is to allow the persistent server process to continue running when the front-end is not running. This feature is important since players may only rejoin a game at a drop-in point. Therefore, for players to make significant progress away from the drop-in points, there must be a process controlling their active(s) which runs continuously. What happens in this case, is that the player gives general instructions to the actives through the front-end describing how they are to behave (for example, “continue towards Mars”). The front-end disconnects and the actives use their internal smarts to follow the instructions given as best they can.

### 3.3 Sphere of influence

Logically we may view the Fleet universe as composed of millions of separate autonomous actives. [Ignore for the moment that some of these actives happen to be owned by the same player and assembled into collections running on the same persistent server.] With the scalability requirements given in section 2, to connect every active with every other active would require  $O(10^{12})$  individual connections (or  $O(10^6)$  connections to each active). It would also require that somewhere there exist global state about every active in the universe, and that state would have to be updated every time any active made a move.

To meet our scalability requirements, we define a sphere of influence around each active. The sphere of influence is a set of actives with which the current active is connected. One may think of the sphere of influence as those actives which the current active can see and/or which might influence the current active.

When talking about the sphere of influence, one must be careful not to get confused about the difference between *physical* network layout and *logical* network layout. The former is simply the topology of the Internet, or whatever networks Fleet runs over. The latter is the topology of the simulated Fleet space. Two actives may be close together in Fleet space, but physically located in England and Australia. Similarly, two players in London, England may be present at opposite ends of the simulated Fleet universe.

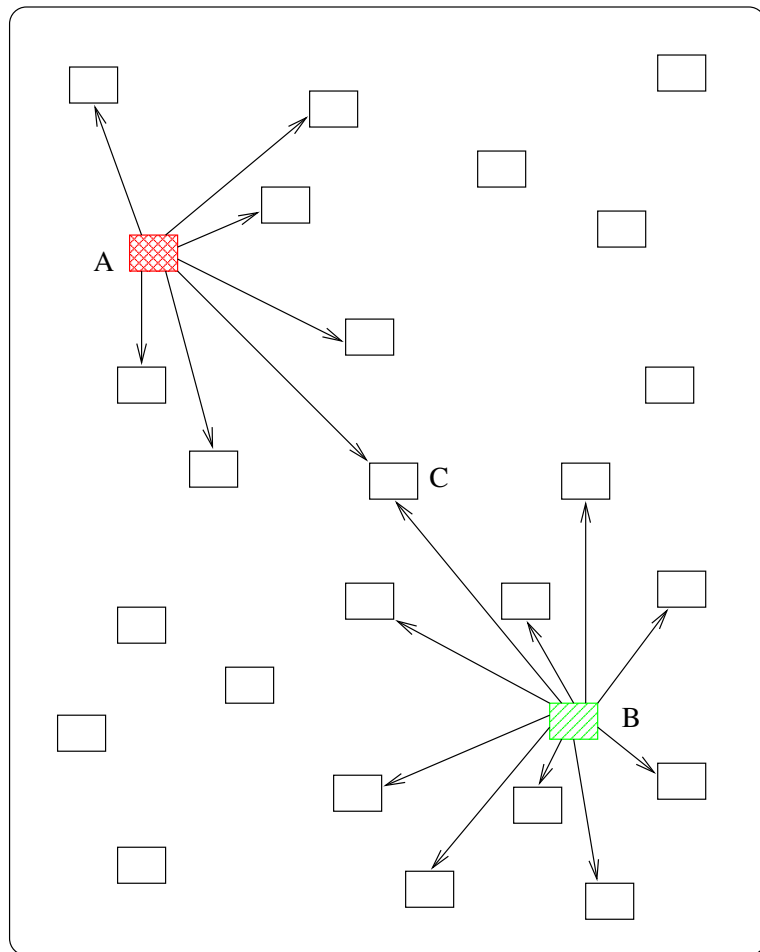


Figure 3.2: Logical network layout showing spheres of influence.

Figure 3.2 shows a logical network layout which two highlighted actives and the spheres of influence of those two actives. The diagram is in 2D for simplicity, but, of course, Fleet is played in 3D space.

### 3.3.1 Notation

#### The influences relationship

The arrow relationship in figure 3.2 we shall write  $X \rightarrow Y$ , meaning “ $X$  **influences**  $Y$ ”. So, in this figure, we have the following relationships of interest:  $A \rightarrow C$  ( $A$  influences  $C$ ) and  $B \rightarrow C$  ( $B$  influences  $C$ ). The figure only shows a few relationships. In reality, every active has its own sphere of influence.

The influences relationship is reflexive and symmetric but not transitive. In practice this means:  $X \rightarrow X$  for any active  $X$  (not a spectacularly interesting statement), and  $X \rightarrow Y \Leftrightarrow Y \rightarrow X$  which is quite a strong and useful condition. It implies in the figure that  $C \rightarrow A$  and  $C \rightarrow B$  although these are not shown.

An influences relationship like  $A \rightarrow C$  entails the following:

- Active  $A$  multicasts all its network messages to active  $C$ .
- Active  $A$  may influence  $C$ , for example, by firing a laser at it, or by exploding near it.
- Active  $C$  polices active  $A$ 's activities.

Conversely, if  $A \not\rightarrow B$ , then the following is entailed:

- Active  $A$  does not send any network messages to  $B$ .
- Active  $A$  may not influence  $B$  in any way. In other words,  $B$  is “unconcerned” with anything  $A$  does.
- Active  $B$  is not involved in policing active  $A$ 's activities in any way.

#### Distances between actives

We write  $d(X, Y)$  to mean the distance (in Fleet space) between actives  $X$  and  $Y$ . The distance is simply computed as:

$$d(X, Y) = \sqrt{(X_x - Y_x)^2 + (X_y - Y_y)^2 + (X_z - Y_z)^2}$$

where  $X$  and  $Y$  have known coordinates  $\begin{pmatrix} X_x \\ X_y \\ X_z \end{pmatrix}$  and  $\begin{pmatrix} Y_x \\ Y_y \\ Y_z \end{pmatrix}$  respectively.

### 3.3.2 Sphere of influence algorithms

There are several related problems connected with constructing the sphere of influence:

- How is the sphere of influence constructed initially? In other words, how do actives find each other and construct a web of influence relationships? This is known as the *bootstrapping problem* and is dealt with last (it is simpler than it sounds).
- What heuristics are used to maintain the influence relationships and change them as actives move around Fleet space?
- Is it possible for actives to subvert the sphere of influence in order, for example, to subvert policing?

The following properties are also desirable:

SOI1 If two actives are close together, then they influence each other.

SOI2 If two actives move far apart, then eventually they stop influencing each other, thus minimizing the number of network connections required.

SOI3 If two actives are sufficiently close together then they influence (mostly) the same actives. This property is desirable to allow the GIVE-TAKE protocol to be policed correctly.

SOI4 The symmetry property of the influences relationship should be maintained.

### 3.3.3 Updating the influences relationship

Revisiting figure 3.2, we see that because  $A \rightarrow C$ ,  $C$  should always know  $A$ 's position. Similarly because  $B \rightarrow C$ ,  $C$  will know  $B$ 's position. Figure 3.3 extends the previous figure showing all actives which influence  $C$  and naming them. We see from this that  $C$  can easily build an accurate database containing the positions of actives  $A, B, D, \dots, K$ .

We work on the principle that the network is slowly flooded with information about the location of every other active in a particular region of space, and that this way, eventually every active will find out about every other active in its area, and be able to decide who should join its sphere of influence.

First of all, as described above, all actives build a database based on their direct knowledge of the positions of their immediate neighbours. They infrequently multicast this information to their sphere of influence in a POSITIONS message.

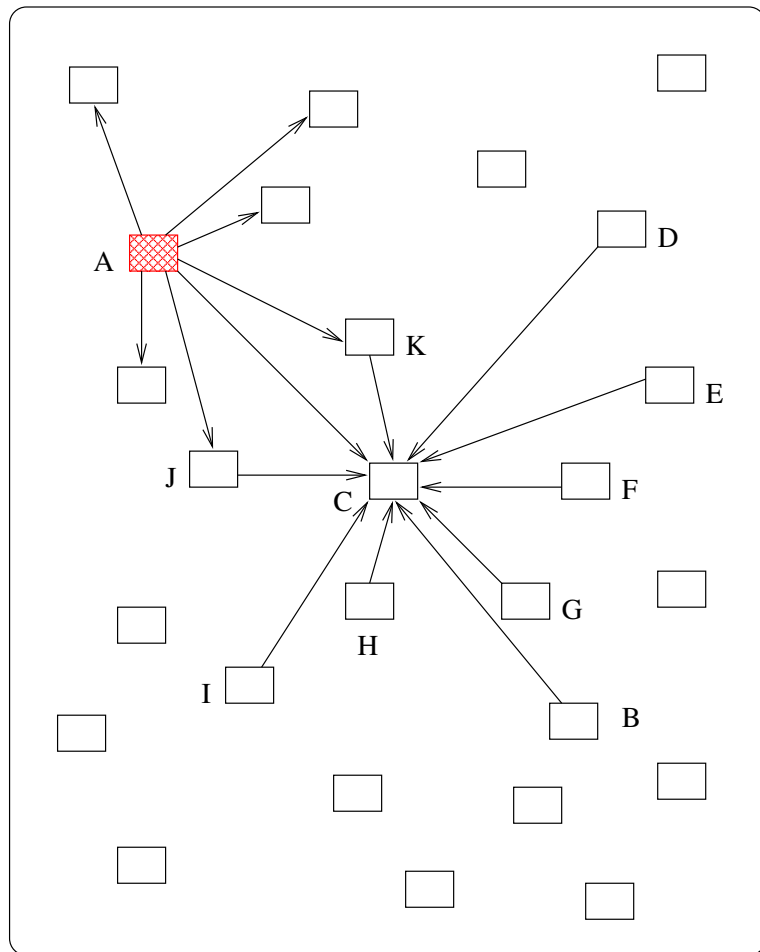


Figure 3.3: The actives which influence *C*.

From receiving other POSITIONS messages, actives are also able to complement their database with indirectly learned knowledge. The indirectly learned knowledge is also retransmitted in POSITIONS messages.

In this way, actives build up knowledge about the actives they are currently directly connected to, and about their neighbours and so on. Eventually a complete picture of the local area can be obtained.

XXXXXXXXXXXXXXXXx

### 3.3.4 Preventing cheating

### 3.3.5 Bootstrapping

An active enters the Fleet universe in two ways: either it enters at a drop-in point, or it is forked<sup>3</sup> off from another active. Both cases are in fact the same, since in the first case the drop-in controller for that drop-in point simply forks the active into the universe. In both cases, then, we have a parent active which forks to create a child active. The child active simply inherits an identical copy of the parent's sphere of influence, and it adds the parent to this set. Actives within the parent's sphere of influence are notified of the creation of the child active, and they add this active to *their* sphere of influence. The reader should satisfy themselves that because the parent and child actives start very close to each other, the invariants given above are maintained.

A rather more subtle bootstrapping problem occurs in the context of creating a completely new *game* of Fleet. How do drop-in controllers get their initial spheres of influence? The answer is simply that a game of Fleet starts with a single super drop-in controller, which we shall whimsically call "God". God starts with an empty sphere of influence, and creates the initial set of actives (sun, planets, space stations, other drop-in controllers and so on) as required. God then retires from the game ...

## 3.4 Multicasting to the sphere of influence

There is only one communications method which Fleet clients need to support: a reliable multicast (one-to-many) protocol from the client itself to all other actives in the client's sphere of influence.

The multicast protocol should obey the following properties:

- M1 The protocol must be reliable. Messages should always be delivered, except in case of network failure when the protocol should detect the failure and inform the application layer.

---

<sup>3</sup>Things will get confusing if the reader is not familiar with Unix process creation and the `fork(2)` system call.

- M2 The protocol should support sending arbitrary sized messages, and should provide a way to separate one message from another.
- M3 Messages must be delivered to the application layer in the same order that they were sent (FIFO ordering). Global ordering is however not required.
- M4 There is only one sender—the current active—and the group of receivers is always equal to the sphere of influence of the current active. The protocol does not need to support any other types of communication.
- M5 It should not be possible for the sender to subvert the protocol and send one message to one group of receivers and another message to another group of receivers (secure). If this is attempted, the protocol should inform the application layer in at least one receiver.
- M6 It should not be possible for one player to forge a message as if it originated from another player. Network messages are always signed, and hence they cannot be forged under any of the protocols.
- M7 The protocol should fail gracefully under heavy network load.

We will initially implement two different multicast protocols, designed respectively for LAN and Internet play. The LAN version is a cut-down multicast protocol based on broadcast ethernet packets, designed simply to test Fleet. The Internet version is the full multicast protocol with knobs on, to be used when Internet-wide Fleet games are being played.

Both protocols obey the M\* properties. The Internet version supports some additional properties designed to make it easier to deploy on the Internet (for example, it knows about proxies, NAT and firewalls, and may use IGMP where available).

### 3.4.1 LAN protocol

The LAN protocol allows small games of Fleet to be played on machines connected by a shared ethernet segment. It uses broadcast ethernet packets, and is not very scalable. However, it is ideal for testing.

The protocol logically comprises two layers:

- 1 A reliable broadcast protocol based on variable-sized UDP packets (some systems will only support 512 byte UDP packets however).
- 2 A variable-sized message delivery protocol built on top of layer 1 which allows messages of arbitrary size to be sent in several packets and, conversely, many short messages to be sent in a single packet.

### Layer 1: Reliable UDP broadcast

The reliable UDP broadcast protocol relies on the fact the ethernet is mostly reliable. We therefore use a negative acknowledgement (NACK) protocol. Such a protocol has almost no overhead when the medium is reliable, but suffers a large penalty when a packet is dropped.

Actives send UDP packets to the broadcast address. Each packet contains a monotonically incrementing **sequence number** and a **transmit counter**, which, first time the packet is transmitted, is always zero. They store a backlog of the packets they have sent so that they can retransmit them if necessary. The length of the backlog is a configurable period of time.

Receivers, listening on the broadcast address also, do not need to acknowledge packets that they receive.

If a receiver gets a packet with a sequence number *larger* than what it was expecting, then it knows that one (or more) packets must have been dropped. It then sends a NACK packet directly back to the sender containing the sequence number(s) and transmit counter(s) of the missing packet(s) that need to be retransmitted. The sender rebroadcasts the dropped packet, with the transmit counter incremented. If a sender has already retransmitted a packet and receives a NACK message for a packet with a lower transmit counter, then it ignores the NACK message for that packet.

Receivers wait after sending a NACK. If the requested packet(s) are not retransmitted after a short period of time, the receiver resends the NACK, increasing the transmit counter by one each time. The frequency of NACKs should back-off exponentially each time to allow the network to fail gracefully.

Since there is a single LAN segment, they should never receive packets out of order from a single source.

UDP has an effective maximum packet size of at least 512 bytes<sup>4</sup>. Less the sequence number (4 bytes) and transmit counter (another 4 bytes) we guarantee to be able to send at least 504 bytes in each packet. On most systems the maximum packet size is much larger (normally 8 or 16 KBytes). The maximum size of packets that layer 1 sends is configurable.

### Layer 2: Variable-sized message delivery

For layer 2 we assume that layer 1 is effectively just a reliable sequenced one-way byte-stream (rather like one half of a full TCP connection). Variable-sized messages simply contain a 4 byte header giving the length of the message, followed by the message itself, length rounded up to the next multiple of 4 bytes. Layer 1 packets are flushed if new messages do not arrive at layer 2 within a certain small period of time.

---

<sup>4</sup>Actually the minimum maximum packet size is just slightly larger than 512 bytes.

### Message properties

- M1 Reliable: Yes (layer 1 retransmits).
- M2 Arbitrary sized packets: Yes (layer 2 can split packets over any number of UDP packets).
- M3 FIFO ordering: Yes (layer 1 sequence numbers).
- M4 Messages reach all members of sphere of influence: Yes, because messages are broadcast to all other stations on the LAN.
- M5 Cannot subvert: Yes? It would certainly be very difficult for a sender to send a well-formed broadcast packet which was only received by certain listeners. It certainly seems strong enough for small games and testing purposes.
- M6 Cannot forge: Yes. Messages are signed.
- M7 Fails gracefully: Possibly. For testing purposes we assume that the network will be lightly loaded.

### 3.4.2 Internet protocol

*Note:* The Internet multicast protocol is much more hairy. I think it should be based on TCP (which is probably better than any reliable protocol we could invent, and is certainly well tested on the Internet!). It must involve proxies to allow it to work through firewalls and between NAT address spaces. There should be some way for the protocol to find a shortest path.

## 3.5 Components and assemblies

Components are Fleet game objects. They are tradable tokens with properties similar to digital cash (e-cash).

### 3.5.1 The Component class

Components are objects arranged into an extensible class hierarchy. All components are created from classes derived from the `uk.co.demon.annexia.fleet.Component` base class. All components contain the following attributes which are coded into the base class:

**class name** The class name of a component is the fully-qualified Java class name as a text string. For example, Fleet money has the class name `uk.co.demon.annexia.fleet.Credit`.

- serial number When the issuer creates a component, he gives it a serial number. This serial number should be unique across all instances of a particular class, so if there are two issuers for a particular component class, they must first agree to split the serial number space between them. Two components are equal if and only if they have the same class name and serial number.
- mass Components have a mass, measured in kilograms.
- volume Components have a volume, measured in cubic meters.
- issuer The issuer’s name.
- expiry date The expiry date of this component. The component must be reissued by the issuer before this date, otherwise it expires and becomes invalid.
- units The number of items that this component represents. Normally this field will contain 1, but it is possible for issuers to create large value components in units of 2, 4, 8, 16, 32, . . . and so on, and, as we shall see below, trading posts are programmed to “give change” for these large units.
- manufacturer Fictional component manufacturer (a string).
- name Fictional component name (a string).

### 3.5.2 Trusted classes

Certain classes of components are so important to the game that we cannot allow any player to issue components of that class. This applies not just to specific classes but to specific parts of the class hierarchy. Most Fleet games are played with an open hierarchy—in other words, players may create new classes of component outside the trusted areas of the class hierarchy. It is possible that some Fleet games may be played with a closed hierarchy. The root page of the game contains bytecode which checks that a component is issued by the correct player.

### 3.5.3 The class hierarchy

There are several special types of component. These are:

- Tradable Components which implement the `uk.co.demon.annexia.fleet.Tradable` interface may be traded (bought and sold) between actives. There is further discussion of trading below<sup>5</sup>.

---

<sup>5</sup>Since just about every component can be traded, this interface is probably unnecessary.

**Housing** Components which implement the `uk.co.demon.annexia.fleet.Housing` interface have a special purpose in life. The housing describes what the active looks like to other players, where its weapons and engines are located and pointing, the volume and shape of the active, and what it is made from. Every active has to have exactly one housing (but may carry other housings in its cargo hold).

**Converter** Components which implement the `uk.co.demon.annexia.fleet.Converter` interface are used when making special network messages informing other actives of their moves and intentions. Fleet has the following types of converters:

**Engine** Converts fuel into forward acceleration.

**Rotational engine** Converts fuel into rotational acceleration.

**Shield** Converts energy into protection against attacks.

**Trigger** Converts explosives into explosions.

**Laser** Converts energy into laser fire.

The actual class hierarchy of Fleet is quite complex. For instance, all types of fuel are in general derived from the `uk.co.demon.annexia.fleet.Fuel` abstract base class. This is merely a convenience. The only components which are treated specially by the Fleet engine are housings and converters. The Fleet engine does not (directly) know anything about “fuel” for instance.

### 3.5.4 The Assembly class

All assemblies are created from classes derived from the `uk.co.demon.annexia.fleet.Assembly` base class. All assemblies contain the following attributes which are coded into the base class:

**class name** As for components, the fully-qualified Java class name as a text string.

**mass** The total mass of all components.

**volume** The total volume of all components.

**components** The list of basic components which make up the assembly. Assemblies cannot be constructed hierarchically (ie. one cannot construct one assembly from several others without first disassembling the others).

**housing** A housing which contains the components and describes them.

**required components** The housing requires that certain components be present, and this field contains a separate list of those components.

**engines** A list of engines fitted to the housing.

lasers A list of lasers fitted to the housing.

manufacturer Fictional assembly manufacturer (a string).

name Fictional assembly name (a string).

Classes derived from `Assembly` also contain smarts which run when the assembly is turned into an active thread (see section 5.1).

Like the `Component` class hierarchy, classes derived from `Assembly` are restricted and controlled by code on the root page. However, unlike classes derived from `Component` anyone can create assemblies (they are not signed). When assemblies are traded, they are checked to ensure a minimum “quality level”, namely that they contain sufficient of each component to ensure a fair trade. Constituent components in an assembly are, of course, signed and checked in the normal way.

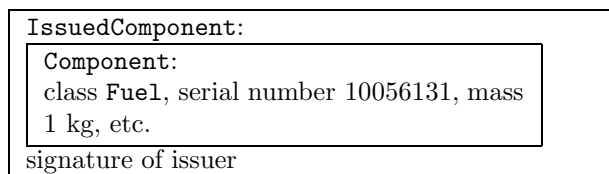
### 3.5.5 Component lifecycle

Components come into life by being issued by certain players. Components can then be traded (bought and sold) between players—directly, or through a trading post. At some point, components are destroyed. Destruction can come about in one of several ways: either because the components are destroyed in battle, or because the components expire and are not reissued, or because the player voluntarily destroys them in order to perform some action (for example, accelerating forwards requires, indirectly, that a player voluntarily destroys fuel components).

Components are fundamental to the game, and it must not be possible for players to issue or duplicate critical components freely. Therefore Fleet employs strong authentication techniques to protect the integrity of components, ensure they are issued by whom they are said to be issued, and protect against double-spending (duplicating) components. The problems are similar to those of digital cash. Some of the discussion that follows overlaps with section 3.8 on policing.

#### Ownership of components and the chain of ownership

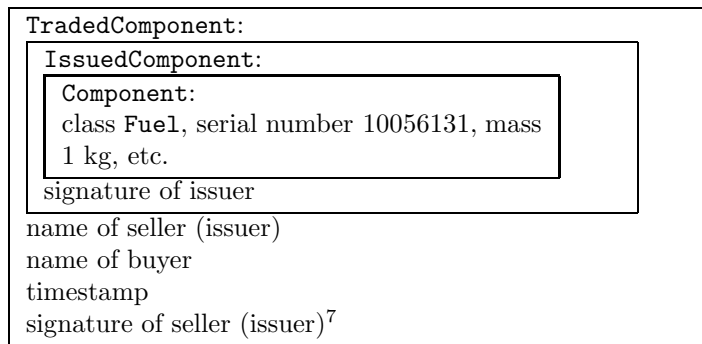
The basic `Component` object describes a component. However, a `Component` object has no intrinsic value unless it is created by a trusted issuer and signed by that issuer. As we will describe below, when an issuer creates a component, they sign it, creating an `IssuedComponent` object. The component (now having an intrinsic value) looks like this:



The signature confers ownership of the component to the issuer. It also ensures that the embedded **Component** object cannot be altered maliciously, and, since signatures are based on strong authentication, means that players cannot forge components.

Components may be traded between players, and trading is usually (though not always) done through trading posts. Trading and trading posts will be described in more detail below. When a component changes hands between players<sup>6</sup>, the seller creates a **TradedComponent** object. The **TradedComponent** object embeds either another **TradedComponent** object or a **IssuedComponent** in the case where the player issuing the component is then going on to sell it.

Therefore, after the issuer of the **Fuel** component above decides to sell it, the component will look like this:



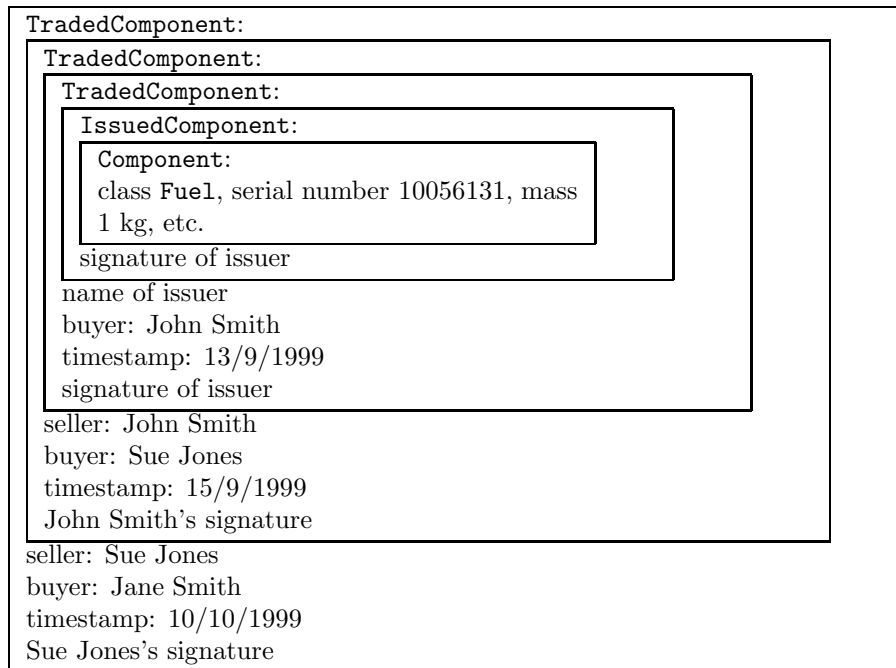
Assuming the transaction is successful, the buyer now has rights and ownership of this component. By signing the **TradedComponent** the seller hands over rights to the buyer.

Of course, none of this addresses the issues of how the component is transferred over to buyer or whether the buyer pays a fair price for the component. Those issues are addressed below.

After a component has been exchanged many times, a so-called **chain of ownership** is established, as in the following example:

---

<sup>6</sup>Trading posts are treated just like other players for the purposes of trading.



What happens if a seller hangs onto the component and then tries to sell the same component again later to someone else? Eventually the double-spent component will be detected, though probably not at the point that the component is traded. We deal with how ownership and double-spend are policed below.

Unlike digital cash, component trading is not anonymous.

Assemblies themselves are not signed or owned. Instead, individual components are signed and owned.

### Issuing components

A player who issues new components is called an issuer. An issuer creates a new component in the following way:

- 1 They create a new instance of the required class.
- 2 They create a unique serial number (must be unique over all instances of that class). If there are two issuers responsible for a particular class of component, they must have agreed beforehand to split the serial number space between them so that they cannot both issue a component with the same serial number.
- 3 They set the issuer field in the component to their own name.

- 4 They set an expiry date for the component. The reason for having an expiry date is explained below. Normally, this is set to the current time plus some fixed number of days, weeks or months. Common components with a high turnover (such as fuel) have shorter expiry times than expensive infrequently requested components.
- 5 They set other fields in the component object: mass, manufacturer and name.
- 6 They set other fields in the object, depending on the component's type.
- 7 They then serialize the final object into an array of bytes, calculate a cryptographically secure hash over the array and sign the hash<sup>8</sup>. The serialized byte array and signature are saved into a new object called an `IssuedComponent`.
- 8 The issuer stores the component class name, serial number and expiry date for the component in a database.

What prevents an untrusted player from just issuing important components such as money? This is where bytecode found on the root page comes into play. All non-cheating players check every component they come across by running it through trusted bytecode found on the root page. This bytecode checks the class name and issuer name to ensure that the issuer is permitted to create instances of the class. The first player who tried to pass off such a component would be treated as a cheat (section 3.8.1). Not all components require trusted issuers. Some components can be freely issued and there is also a place in the component hierarchy for players to create their own trusted components.

When components are issued, they contain an expiry date. It is the responsibility of the owner of the component to go back to the original issuer of the component and have the component reissued (for free) as it nears its expiry date. The expiry date is simply a mechanism to control the number of components in circulation, particularly where people are issued components but fail to use them (because they have lost them or just stopped playing the game). The expiry date can be controlled by issuers to balance the size of the database they must maintain and the network load caused by people getting components reissued. The Fleet persistent server is designed to avoid reissue where possible (by, for example, preferring to burn fuel with the shortest expiry time) and to transparently reissue components before they expire. Thus players should not need to worry much about expiry dates.

---

<sup>8</sup>By “sign a hash”, we mean they decrypt the hash using their private key. The result of this decryption is the signature.

## Trading components

We have already explained how components are “owned”. This section discusses the actual mechanics of trading components.

Any active may give any component or assembly that they own to any other active by the following method. We shall call the active giving the component the “donor” and the active receiving the component the “recipient”.

- The donor and recipient must be within a certain (small) distance from each other. Actives in the sphere of influence of the donor and recipient police the gift and will consider it cheating if the donor and recipient are too far apart.
- The donor ensures (by some other method outside the protocol) that the recipient knows about the gift and is ready to receive it.
- For single components, the donor wraps the component in a **TradableComponent** object, in the process recording the name of the donor, name of the recipient, timestamp, and signing the result. For assembly, the donor does the same operation to each base component in the components list.
- The donor places the **TradableComponent** or **Assembly** in a GIVE message, addressed to the recipient, and sends this message.
- The recipient, if they accept the gift, responds with a TAKE message.
- The donor deletes the component (assembly) from their database to ensure that they aren’t accused of double-spending the same component (assembly) twice later.
- The recipient unwraps the layers of the chain of ownership, checking them. For assemblies, it does this for each base component. It adds the **Component(s)** and the fully wrapped **TradableComponent** or **Assembly** into its database. It will need one of these objects again when it tries to sell on the component (or destroy it), but it is convenient to deal only with unwrapped components internally.
- Other actives in the sphere(s) of influence police the move (see section 3.8).

Trading is built around this basic **GIVE-TAKE protocol**. We first discuss problems with the GIVE-TAKE protocol which make it unsuitable for trading as it stands:

- The protocol does not ensure that the donor receives anything in return for their gift. The recipient might simply run off with the gift.

- Even if we can ensure that the recipients are honest, the protocol is cumbersome because it requires individual buyers and sellers of particular components to find each other, argue about a rate of exchange, and finally exchange the components.
- The protocol does not specify any way to decide on a fair market price for each component.
- The protocol does not allow a large unit component to be split up into smaller units, equivalent to the problem of getting change from a £10 note.

Fleet solves these problems by using **trading posts**. A trading post is a marketmaker offering to buy and sell individual components at fixed prices. A trading post deals in many different types of components and assemblies and generally only exchanges items for cash. Trading posts make a profit on the margin between the price they buy components and the price they sell components. Trading posts are smart actives which adjust their prices based on supply and demand, and to some extent, trading posts compete with each other (if they are located geographically close at least).

Trading posts also offer other services, such as banking, insurance and stock markets, but these are peripheral to their main purpose.

Some trading posts are trusted—in other words they are guaranteed by the owner of the game. These trading posts are listed in the root page. However, there is nothing to stop any player setting up a trading post. These secondary, untrusted trading posts work on reputation alone and there may be no comeback if they rip you off in a deal.

Trading posts advertise so-called **bids** and **offers** for a range of components<sup>9</sup> that they can trade in. The bid is the lower price for the component and it is the price (in credits) at which the trading post will buy units of the component. The offer is the higher price for the component and it is the price at which the trading post will sell units of the component. Following the stock market terminology, the difference between the bid and offer prices is the **spread**.

Trading posts periodically multicast their prices to actives within their sphere of influence<sup>10</sup>. They send out PRICELIST messages, listing components and bid and offer prices.

A player who wishes to trade sends back a STARTTRADE message. A STARTTRADE message contains:

- The class name of the component.

---

<sup>9</sup>I will use “components” in this section to refer to assemblies also.

<sup>10</sup>For policing purposes any active may be a trading post, but there is a limit to how often trading posts may broadcast their price lists. The limit depends upon whether they are trusted (listed in the root page) and how many components they are advertising. This is explained in section 3.8.

- The number of components to trade.
- Whether the player wishes to buy or sell the components.
- The trading price.
- A unique identifier which is used to refer to this particular deal in future messages.

The player can offer any trading price they wish, but in general it only makes sense to send the bid price (if the player is selling) or the offer price (if the player is buying) from the most recent PRICELIST received. The trading post now decides whether or not to take up the player's offer. The trading post is entirely free to reject the offer even if the price is right and the trading post has adequate cash or components to cover the trade. Similarly a trading post may—at its discretion—accept an offer which is way above or below the true market place. The trading post returns either an ACCEPTTRADE or REJECTTRADE message. This message contains the same unique identifier as the STARTTRADE message.

If the trading post sends a REJECTTRADE message, then that particular trade is over. The player may make a subsequent offer if they wish. If, however, the trading post sends an ACCEPTTRADE message, then the trading post commits itself to buying or selling the named components in the agreed quantity at the agreed price<sup>11</sup>.

After receiving an ACCEPTTRADE message, the player gives the agreed number of components or sum of cash to the trading post (using the GIVE-TAKE protocol outlined above) and the trading post gives cash or components back to the player in return. Note that trusted trading posts are underwritten by the owner of the game, and so there is no problem if they go back on the deal. Untrusted trading posts trade on their reputation not to rip off the player.

The trading post is also able to give change for components. For example, a player may wish to sell 7 grams of gold, but own a single 16 gram component of gold. In this case, the player strikes a deal in the usual way for the 7 grams of gold. The player then proceeds to send the 16 gram component (transferring ownership of the whole component to the trading post). Normally the trading post will send back the cash for the 7 grams, an 8 gram gold component and a 1 gram gold component. If the trading post does not have the correct change available, then the trading post will send back the whole 16 gram gold component. The player can infer from this that sufficient change was not available. Other components, including credits, are handled in a similar way.

---

<sup>11</sup>Nevertheless ACCEPTTRADE messages are only valid for a short period of time. The player may back out of a deal at this point simply by not responding to the ACCEPTTRADE message which will eventually lapse.

## Destroying components

Many components are eventually destroyed. Components are destroyed in one of three ways:

- 1 Actives need to destroy components in order to perform various manoeuvres. For example, accelerating forwards requires the active to (indirectly) destroy fuel components.
- 2 When an active is destroyed, all of its components are destroyed too.
- 3 A player may voluntarily destroy components at any time.

There is, in fact, another method by which components can disappear from the game. Components are created with an expiry date, and in the absence of other actions, components simply cease to be valid after they have reached their expiry date. We don't cover this here, since components which expire do so automatically, without requiring any interactions between the current owner of the component and the original issuer.

Destruction of components is unusual in that it is normally done by actives in the sphere of influence in their role policing—rather than being done directly by the active which owns the component. This makes a lot of sense since the active at the centre of the sphere of influence would generally quite like to hang on to the components, and not have them destroyed at all. So it is up to the actives which police to destroy components.

To reduce confusion, we refer to the active which currently owns the components as the “central active” and the actives in its sphere of influence as the “policing actives”.

In case (1), whenever the central active wishes to change course, fire a laser etc., it multicasts a network message containing a list of components to support the move. The policing actives must implicitly destroy the components in this list.

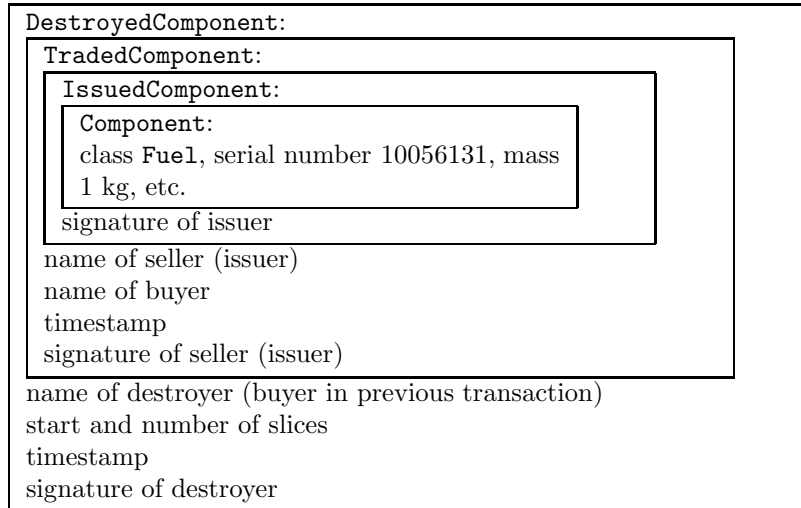
In case (2), we have the case where the central active comes under attack and cannot issue valid PROT messages to resist the attack. In that case, the central active and all its components are destroyed. The central active is required to issue a DESTROY message containing a complete list of the components which it owns. Policing actives can check that the message is both sent and complete (since they already have a database containing all components that the active *ought* to contain), and thus detect cheating.

Case (3) is slightly different (and very unusual). In this case the central active multicasts a DESTROY message containing a subset of the components it owns. The policing actives then go through that list and destroy them.

In all cases we arrive at the situation where all the policing actives surrounding the central active have a duty to destroy a list of components. From a policing point of view, we have to avoid the case where a policing active

(cheater) decides to maliciously destroy components legitimately belonging to another player. The protocol is designed to protect against this case also.

To destroy a component, the player wraps the component in a `uk.co.-demon.annexia.fleet.DestroyedComponent` object and signs it. A component which has been issued, traded once and destroyed would look like this:



*Note:* The start and number of slices fields are explained below.

It is these `DestroyedComponent` objects which are actually sent in network messages and in DESTROY messages.

The idea is that eventually destroyed components migrate back to the original issuer of the component. The issuer is responsible for policing double-spend, and to do this, it needs to see destroyed components (this is explained in section 3.8). The issuer also removes destroyed components from its database. Policing actives, however, do not need to deliver all destroyed components they come across back to the issuer. First of all, there are many policing actives for each central active, and so many copies of each destroyed component are present. Secondly, to do adequate policing, it is only necessary for a certain proportion of double-spent components to be detected. Since the penalty for double-spend is very severe (an instant ban from the game), the issuer only needs to detect reliably, say, 1 in 10 double-spent components for it to be a sufficient deterrent. For these reasons, policing actives only keep a small proportion ( $p_{destroyed}$ ) destroyed components. The other destroyed components are just dropped. This number can be calculated from the average number of actives in a sphere of influence and various other factors.

Policing actives store  $p_{destroyed}$  of the destroyed components they come across in a database. There are certain actives mentioned on the root page as destruction posts. The next time a destruction post enters the active's sphere

of influence, the active uploads its database of destroyed components to the destruction post. To do this, it sends the destruction post a DESTROY message. The destruction post forwards the destroyed components back to the issuer using methods which are outside the scope of the Fleet game.

### Partially destroyed components

It is also possible to partially destroy a component. To see why this is necessary, consider the case where the smallest possible change in rotation or acceleration requires a single fuel component. A simple manoeuvre could require ten or so fuel components to be destroyed. A long haul flight between distance planets might require millions of fuel components.

To reduce the number of certain basic components that actives must carry (and as a consequence reducing the size of many network messages), we allow some components to be **partially destroyed**. Components are split into  $N$  (a large number<sup>12</sup>) of **slices**. When a component is destroyed, the start and number of slices fields are set to indicate which slices are being destroyed. For instance, to destroy the whole component, start is set to 0, and number to  $N$ .

The start and number define a range of slices, and once that range is destroyed, the same range or an overlapping range cannot be destroyed in a subsequent message. Hence, for example, if  $N = 10,000$ , then you could destroy the following sequence:

start	number
0	500
500	2500
3000	2500
5500	2500
8000	500
8500	500
9000	500
9500	500

Players can destroy slices in any order, provided the ranges destroyed do not overlap.

Only components used in status messages (section 3.7) can be destroyed. This basically limits partial destruction to fuel and energy<sup>13</sup>. Double-spend is policed in the same way as for complete components.

Notice that in order to police double-spend accurately with partial destruction, we require the range of slices method outlined above, rather than just allowing the active to declare a single number representing the proportion of the component destroyed. The reason is because issuers do not necessarily see all destroyed parts of a component, and this would allow cheating actives, with

<sup>12</sup>Probably between 10,000 and 1,000,000. The exact of number will be decided later.

<sup>13</sup>It does not make sense to allow explosives to be partially destroyed.

reasonable safety, to destroy a few percent more slices than are present in each component.

### **Reissuing components**

*Note:* This section will contain information on how components are reissued.

### **Components present in an active**

An active consists of the following components:

- housing Every active must hold a housing component. An active may carry several housing components, but the active must declare exactly one of these housing components as its housing.
- required components The housing describes the shape and size of the active—in other words, how it appears to other actives, how large it is, how much cargo it may carry and so on. It also defines certain static components which must be present, and logically constitute part of the housing. Common static components which the housing may require are a certain mass of steel (from which the active is constructed), quantities of high-tech (intended to represent computers and so forth present on-board the active) and plant life (if the active is a space station, then it requires a complex set of life-support technologies including plants to generate food and oxygen). Space stations frequently have very large numbers of static components. They may consist, for example, of thousands of tonnes of steel. When an active is policed, trusted bytecode in the housing is called which verifies the presence of the right quantity of each type of component in the active. Housings are often parameterized objects, and can change their shape and size depending on the number of static components present. For example, to increase the volume of the cargo hold, some housings may allow the owner of the active just to add steel (an unparameterized housing would require the player to drop out of the game to get another housing fitted).
- engines and lasers Housings also define fixtures for optional components such as engines and lasers. If the active wants to use an engine or laser, then it must first fix it into a particular place in the active (and it can't move the component once fitted without a major refit which involves dropping out of the game temporarily).
- cargo In addition, the housing defines a certain volume which can be used to store any components up to that volume. All other (non-static) components are stored here, including fuel to power the engines, shields energy to power the shields and other general cargo.

Actives must declare all components they carry to all other actives in their sphere of influence. All actives therefore build a database of the components carried by actives in their neighbouring region.

## 3.6 Housings and collision detection

In this section we describe in considerably more detail how actives specify their size and shape to others, and how we can detect when two or more actives collide.

### 3.6.1 Size and shape example

In most games of Fleet, housings are trusted components, and the administrator will generally create a few different types of housings, suitable for space craft, cargo carriers, missiles, space stations and so on. Players will usually be permitted to customize these housings to a certain degree.

To give a concrete example: the space ship class of housing does not specify a size or shape, but leaves this information to the user of the class. The user of the class supplies a list of vertices, surfaces and texture maps. However, the space ship housing (being trusted bytecode) polices itself to ensure that:

- The correct amount of steel is supplied to “build” the space ship, given the apparent size of the space ship.
- The cargo volume available again depends on the apparent size of the space ship.
- The space ship is visible to other actives.

Thus players may build space ships with very large cargo bays, but the penalty they pay is that such space ships will be very large (offering a bigger target to attackers) and will require a lot of steel to build (and hence will be heavier, making them more difficult to manoeuvre and more costly to fly).

This section of the document does not discuss this policing any further. The reason is that it is up to the administrator of the game to ensure that the available housings “do the right thing”. The core Fleet game engine does not police housings at all.

### 3.6.2 The Housing interface

As far as the Fleet game engine is concerned, a component which has the `Housing` interface must publically export the following attributes which are used to decide the size and shape of the active, and are also used to perform collision detection. Note that the interface is heavily geared towards being able

to rapidly display actives in OpenGL, since OpenGL is the 3D graphics library of choice on all Fleet platforms.

**display description** The display description field is a GL-oriented list of vertices, surfaces, lines and textures which together describe exactly what the active looks like. The display description may be limited in size—again, not by the Fleet game engine, but by the housing classes themselves—or it may contain alternative representations so that slower machines might display, for example, just a wireframe of a particularly complex structure.

The origin of the display description should be located somewhere near the logical centre of gravity of the active (this is not a requirement, but makes the maximum radius field make more sense).

The axes of the display description are measured in meter units.

If the housing has a “forwards” direction, then traditionally the display description has the forwards direction pointing towards greater Z values. However, this is not a requirement, nor is it enforced by the Fleet game engine.

The front-end builds the display description into a GL display list as early as possible to allow the active to be displayed as rapidly as possible. Animated textures are permitted, but it is not possible for actives to change shape (without dropping out of the game for a “refit”). One possible extension to Fleet for the future would be to allow jointed sections of actives to move around. This would require considerable additional work on the Fleet protocols.

**max. radius** The maximum radius field describes the radius of the smallest possible sphere centered at the origin which completely contains the active. This field is used when doing collision detection. Only if two neighbouring actives have overlapping spheres does the Fleet engine compare the boundaries to work out if they have collided.

**boundaries** The housing declares a list of simple polygons built into a CSG (constructive solid geometry) tree using union, intersection and complement operators. The boundaries are used to accurately compute collisions and laser hits.

The origin and scale of the axes of the CSG tree must be the same as for the display description. Other than that, the display description and boundaries are completely separate and unrelated objects. It is up to the housing to ensure that the boundaries and display description coincide<sup>14</sup>.

---

<sup>14</sup>Roughly speaking the boundaries should describe the extent of the space ship as closely as possible, modulo speed factors (simplifying the boundary description will probably considerably reduce time taken to compute collisions) and complexity.

fittings Housings may offer places to fit engines and lasers. If they offer these fittings, then they offer a list of these fittings here.

policing Housings supply methods to police the list of required components.

### 3.6.3 Collision detection, laser hits, explosions

*Note:* This section will describe how collision detection works.

### 3.6.4 Refitting an active with a new housing

*Note:* This section will describe how you need to drop out the game to drop out of the game to refit with a major new housing or to move lasers or engines. However, cargo changes probably don't require you to drop out of the game. Perhaps, however, there ought to be a way to refit in flight (with some associated penalty of course).

## 3.7 Status messages

Fleet actives announce their intention to **move** by multicasting one of a number of **status messages** to their sphere of influence.

With traditional multi-player games, status messages simply contain a command like "I am moving here" or "I have fired my laser". In Fleet, status messages contain important extra information, known as the **justification**. The justification is, if you like, proof that the active is permitted to make the associated change in status. The other actives in the sphere of influence check the justification field to detect cheating. The policing is described in section 3.8.4. This section describes the contents of the status messages themselves.

### 3.7.1 The ACCEL message

The active declares its intention to accelerate along a particular direction over a set period of time. Actives are only permitted to accelerate in the direction that their engine(s) are pointing<sup>15</sup>. Table 3.1 lists the fields present in the ACCEL message.

### 3.7.2 The ROT message

The active declares its intention to accelerate or decelerate its rotation along a particular axis of rotation. Actives need a rotation engine to be able to rotate, but unlike a normal engine, rotational engines allow the active to rotate in any direction they wish. Table 3.2 lists the fields presents in the ROT message.

---

<sup>15</sup>Note that actives that wish to accelerate *and* decelerate require two engines pointing in opposite directions, just like in real life!

Field	Description
Time	Start and end times for the acceleration.
Direction	Direction for the acceleration.
Magnitude	Magnitude of the acceleration.
Engine	Engine component used to perform acceleration, and the fuel (or other) components destroyed by this engine.

Table 3.1: Fields in the ACCEL message

Field	Description
Time	Start and end times for the acceleration.
Direction	Direction for the acceleration.
Magnitude	Magnitude of the acceleration.
Engine	The rotational engine component used to perform acceleration, and the fuel (or other) components destroyed by this engine.

Table 3.2: Fields in the ROT message

Field	Description
Time	Start and end times.
Magnitude	Magnitude of protection.
Shield	Shield component and the energy (or other) components destroyed by this shield.

Table 3.3: Fields in the PROT message

### 3.7.3 The PROT message

An active sends out protection messages when it comes under attack by laser, explosion or fire. Protectional messages require a shield converter to be present, and most shield converters consume energy components. When an active comes under attack, it must (within a short period of time owing to latency) start sending out sufficient PROT messages to cover damage. If an active does not or cannot send out sufficient PROT messages, then the active is immediately destroyed<sup>16</sup>, and must send a DESTROY message instead, containing a list of all its components, destroyed.

Table 3.3 shows the fields present in a PROT message.

### 3.7.4 The EXPLODE message

An active creates an explosion at its current location. The explosion affects the active itself (usually destroying the active) and all actives within a certain range. The destruction that an explosion causes is affected by the magnitude of the explosion and the distance of other actives from the epicenter.

Table 3.4 lists fields present in EXPLODE messages.

### 3.7.5 The LASER message

An active fires a laserbeam from one of its onboard lasers in a particular direction. The laserbeam damages everything in its path in proportion to the magnitude of the beam and the distance of the other active from the laser itself.

Table 3.5 lists fields present in LASER messages.

---

<sup>16</sup>There is no concept of “percent damage” in Fleet. This is essentially because in Fleet actives cannot really have any persistent state that is not represented in one way or another by components. There is no mechanism in Fleet to administer or police a “percent damage” field attached to an active. This works well for small to medium sized actives. However, it means that a large active, such as a space station, could, in theory, be destroyed by a single shot if it didn’t have a shield. Large actives such as space stations are expected to carry a shield and *a lot* of energy to power the shield. In addition, space stations can avoid damage by fighting back, with both fixed lasers and armies of space ships which destroy any attackers.

Field	Description
Time	Start and end times.
Magnitude	Magnitude of the explosion.
Trigger	The trigger converter used to create the explosion and the explosive (or other) components destroyed by the trigger.

Table 3.4: Fields in the EXPLODE message

Field	Description
Time	Start and end times.
Magnitude	Magnitude of the laser.
Laser	The laser converter used to create the laserbeam and the energy (or other) components destroyed by the laser.

Table 3.5: Fields in the LASER message

### 3.7.6 The FIRE message

*Note:* The FIRE message was a good idea, but doesn't have a natural converter. Maybe drop it later?

### 3.7.7 The FORK message

The current (parent) active “forks” a new (child) active very close to the parent. Normally the child active is constructed out of components from the parent.

The child active has to be created close to the parent and with a similar position and momentum.

Table 3.6 lists fields in the FORK message.

Field	Description
Time	Time of the fork.
Status	Complete status of the child active.

Table 3.6: Fields in the FORK message

### 3.7.8 The JOIN message

*Note:* The JOIN message is a proposed extension to the Fleet protocol, which would allow two actives close together, owned by the same player to join together to form a single active. In practice, one of the actives would just disappear, and its components would be handed over to the other active. This could be useful in one particular situation: where a heavy space ship contains a detachable lightweight section. The detachable section breaks off from the heavy section during battles, and afterwards the sections rejoin. It may be possible to simulate a JOIN message using the GIVE-TAKE protocol however (the child active just gives all of its components to the parent, and then the child ceases to exist). This would only require changing the semantics and policing of the GIVE-TAKE protocol slightly, and so it might be preferable to implementing a whole new message type.

### 3.7.9 Network latency and timestamps

Fleet is an asynchronous game running on distributed machines, and so unless clocks are synchronized across machines there could easily be disputes about when a particular event occurred. Therefore we require that all participating machines have access to clocks synchronized with UTC<sup>17</sup>. For Unix clients this is not usually a problem, since `ntpd` is widely available and widely used. For other clients, code is included in the server to ensure that a reliable timesource is available<sup>18</sup>. Note that it is vital that clocks on all machines are synchronized to within a few milliseconds of each other (easily achievable with NTP) and machines which are not synchronized this closely will not be able to even join the game.

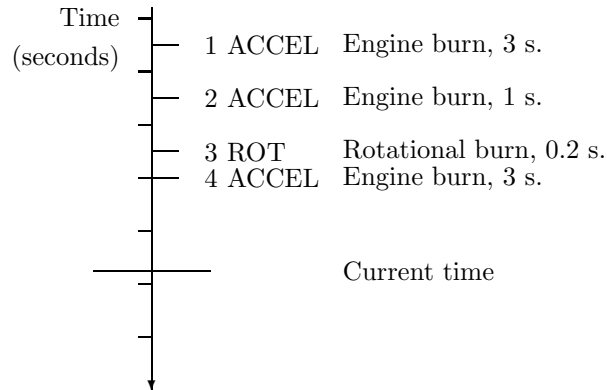
All network status messages are stamped with the absolute time (UTC seconds and milliseconds from the Unix epoch) that they occur. When another server receives a message, it is also timestamped again as soon as it has been received. This second timestamp allows network latency to be measured and policed. Since actives use a protocol which is FIFO ordered and reliable, it is cheating for an active to send one message which appears to have originated earlier than a previous message. Apparently large jumps in network latency may also be forbidden.

With reliable timestamps attached to each message, servers build up a picture of other actives which looks something like this:

---

<sup>17</sup>Note that UTC is the “real time”, and Fleet never uses local time, partly because players next to each other in the Fleet universe can easily be in different timezones.

<sup>18</sup>Initially elaborate schemes were devised to synchronize clocks and measure network latency, but such schemes are simply unnecessary if we assume that NTP is available, and is simpler just to require this.



Here we see that four messages have been sent so far by this active, at times  $t_0 + 0.5$ ,  $t_0 + 1.5$ ,  $t_0 + 2.5$  and  $t_0 + 3$  seconds respectively. The current time is just before  $t_0 + 5$ , but we do not have the “full story” of this active up to this time yet. Because of network latency a fifth message could easily appear between (but not before)  $t_0 + 3$  and the current time. Therefore the “current position of the active” maintained in the server is based on estimation from the point of the previous message received up to the current time.

Some points to note:

- Actives do policing only once the concrete facts of the situation are known. That is to say, in the example above policing can only be carried out on the active’s state up to and including message 4. Policing which requires an active’s exact position to be known (ie. policing LASER, EXPLODE and PROT messages) is therefore done some seconds after the event. Actives are required to send out something every so often (at least an empty, timestamped heartbeat message) so policing is always done within a bounded amount of time.
- Different estimation algorithms can be used in different implementations of the Fleet client, but the supplied estimation algorithm is “dead reckoning”—a very simple algorithm which just assumes that no more messages will be received between the last message and the current time. There are more advanced methods based on second- and higher-order estimating functions.

## 3.8 Policing

Fleet’s peer-to-peer architecture dispenses with the central server. It’s open sourcecode allows players to modify their clients as they like. In ordinary commercial multiplayer games (MPGs) players are policed either by the central

server (which knows, for example, how many missiles they have fired and how many they have left) or by the fact that the source is closed (and hence it is not simple for non-hackers to change, for instance, the number of missiles they start with). These assumptions do not apply to Fleet, and so we must implement an explicit, complete and strong way to police players. Policing applies in spite of modifications that players can make to their own clients.

In Fleet, it is the responsibility of actives to police each other. One active is policed by all the actives within its sphere of influence. Policing has two aspects:

- An active can make one out of a finite set of “moves”. Each move is multicast to the other actives in the sphere of influence, and each other active polices the move.
- An active should not be able to manipulate its sphere of influence or alter the multicast protocol itself. Other actives are responsible for policing the sphere of influence rules and the multicast protocol.

### 3.8.1 Cheating

What happens if one active detects that another active is cheating? One might use various formal and automatic schemes to report offenders to a central authority. Ironically, these schemes can easily be abused themselves. In Fleet, as soon as one active detects another active cheating, the active displays a message containing the cheater’s name (which is normally then printed out by the front-end). The active then drops the cheater from its sphere of influence and refuses to send or reply to network messages from it for a period of time<sup>19</sup>. Since all actives surrounding the cheater act the same way, this in effect means the cheater drops out of the game and must restart at a drop-in point. Persistent cheaters will be reported by players informally back to the authority running the game, by mail and news, and, if they become notorious enough, they will be added to a permanent **blacklist** available on the root page.

Players detected cheating by the authority running the game are likely to be added to the blacklist immediately. The most important case where this happens is when the authority is responsible for issuing components, and detects players double-spending.

### 3.8.2 Trusted actives

Certain actives are trusted, and do not need to be policed. There are two classes of trusted actives:

- Actives which are present in the same persistent server must be owned by the same player, and so checking is not required.

---

<sup>19</sup>This period of time doubles each time the cheater is detected cheating.

Capability	Description
Well-formed	The message itself is well formed.
Components	The engine and the destroyed components it contains are well formed. The destroyed components are added to the database of destroyed components.
*-Converter	The engine is a component of the active, and is fixed in the active's housing. The engine passes its own correctness test (ensuring that enough fuel components have been passed). Engine does not exceed its conversion rate.
*-Checks	Magnitude of acceleration is correct considering mass of active and power produced by engine and fuel components. Direction of acceleration matches direction of engine.
Time	Start and end time are reasonable.

Table 3.7: Checks performed for ACCEL and ROT messages.

- Actives which have **capabilities** assigned to them by a trusted authority (from the root page). Each policing step has an associated capability. Actives can be assigned capabilities from the root page which allow them to avoid being policed for particular steps. For example, the FORK message requires two main checks: that the child components come from the parent, and that the child is created close to the parent. Drop-in controllers are trusted and listed on the root page. They avoid the first check, but not the second.

### 3.8.3 Policing the sphere of influence

*Note:* This is a tricky area . . .

### 3.8.4 Policing status messages

#### The ACCEL and ROT messages

When a policing active receives an ACCEL or ROT message, it performs the checks outlined in table 3.7.

Capability	Description
Well-formed	The message itself is well formed.
Components	The shield and the destroyed components it contains are well formed. The destroyed components are added to the database of destroyed components.
Prot-Converter	The shield is a component of the active. The shield passes its own correctness test (ensuring that enough energy components have been passed). The shield does not exceed its conversion rate.
Time	Start and end times are reasonable.

Table 3.8: Checks performed for PROT message.

### The PROT message

When a policing active received a PROT message, it performs the checks outlined in table 3.8.

### The EXPLODE, FIRE and LASER messages

Explosions, fires and laserbeams are policed in much the same way as ACCEL and ROT messages. Table 3.9 lists the checks performed.

### The FORK message

The FORK message is easiest to police. Creating a new active mainly involves creating new local objects containing the state of the child, copying components from the parent to the child and adding the child to the sphere of influence. Table 3.10 lists the checks performed.

## 3.8.5 Policing components

*Note:* This section will discuss how components are policed for double-spend. A brief summary: (1) Actives have to declare their complete component list. I'm not sure yet whether they will have to declare the fully wrapped components directly, or whether they will just pass out `Component` objects. In the latter case, the policing actives could then query the actives for a particular fully wrapped component. This might reduce network traffic a lot. In either case, policing actives send a small random proportion of their neighbours' components (fully-wrapped) back to the issuer to check. This requires a new CHECK message. (2) Destroyed components are returned to the issuer too. (3) The issuer, by seeing a

Capability	Description
Well-formed	The message itself is well formed.
Components	The trigger/laser and the destroyed components it contains are well formed. The destroyed components are added to the database of destroyed components.
*-Converter	The trigger/laser is a component of the active, and (for lasers only) is fixed in the active's housing. The converter passes its own correctness test (ensuring that enough explosive/energy components have been passed). The converter does not exceed its conversion rate.
*-Checks	Magnitude of explosion/laserbeam is correct considering power produced by trigger/laser components and destroyed components. Direction of laserbeam matches direction of laser.
Time	Start and end time are reasonable.

Table 3.9: Checks performed for EXPLODE, FIRE and LASER messages.

Capability	Description
Well-formed	The message itself is well formed.
Component-checks	Child components come from parent.
Position-check	Child active is created sufficiently close to the parent and with a similar momentum.
Time	Fork time is reasonable.

Table 3.10: Checks performed for FORK message.

fair selection of destroyed and undestroyed components from around the game, can detect double-spent components reliably, and also detect who double-spent them. The double-spending cheater is immediately suspended by adding them to the blacklist. (4) Policing actives check that components destroyed by actives actually belonged to the actives in the first place. (5) Components are checked for correct issuers and within expiry dates.

# Chapter 4

## Tools

### 4.1 Languages

The back-end persistent server process is required to send complex objects over network connections. Because Fleet must be extensible without changes to the common code base, some of these objects passed over the network will contain code as well as data. This code must run in a protected sandbox.

Furthermore, the Fleet persistent server supports several actives which run concurrently as separate threads. Therefore we require a language with good, simple thread support.

The Fleet persistent server design is strongly object-oriented.

We have chosen the Java language to implement the Fleet persistent server. Most of the server will be compiled to native code for speed.

The front-end need to run quickly (above almost all other concerns). It needs to access the OpenGL libraries used to talk to the display.

We have chosen the C language to implement the front-end. Certain parts of the front-end may, after profiling, have platform- and architecture-specific variants written in assembler.

### 4.2 Libraries

The back-end persistent server will be a 100% pure Java program. It will use the JDK 1.2.

The front-end, written in C, requires a high-quality cross-platform 3D library, a portable network API and various miscellaneous operating system functions.

For 3D output and input, we avoid single-platform APIs such as Direct3D, and instead choose OpenGL 1.2 and GLUT. This library is proven for writing

games (GLQuake, for example) and portable across all variants of Unix and Windows.

The network API is the common subset of BSD sockets and Winsock 1. Several macros are needed to, for example, deal with the strange way that Winsock handles error codes.

A simple library will be written which will encapsulate common POSIX.1 file and OS functions across Unix and Windows.

### **4.3 Network**

**Part II**

**Detailed Design**

## Chapter 5

# Persistent server

In Fleet, each player runs their own server (the word “server”, used loosely in this sense, of course), and that server is responsible for managing their own actives.

Actives are divided broadly into two types: those that are entirely driven by their on-board smarts—missiles, for example—and those that are usually meant to be driven by an interactive front-end, but contain smarts to take over when the front-end is not there. Space ships are a good example for the latter type, since we assume that most players will not want to play Fleet  $24 \times 7$  for ever. Fleet does not rigidly impose any structure on actives, but instead allows an active to run any code it likes in a thread in the persistent server and manage communication separately with a front-end (or not) as it likes.

The persistent server also runs threads to handle network communications, policing, and managing the status of the other actives. This allows active threads to run unhindered by network delays, and allows us uniquely to assign a priority to policing, so that on a slow machine, players can limit the amount of time spent policing other actives and increase the amount of time spent actually playing the game.

Figure 5.1 illustrates the block architecture of the front-end.

It is worth spending some time examining this in overview. At the top of the diagram, we show four active threads running. We can assume that the second from left thread in the diagram is running the player’s space ship. It is connected directly to front-end, and is at various times under the control of this front-end and running its own smarts. The other three threads, we presume, are missiles or some other independent actives owned by this player and launched from the space ship in the past.

All active threads will contain a main `runActive` method which has the following form:

```
public void runActive (Active myActive,
```

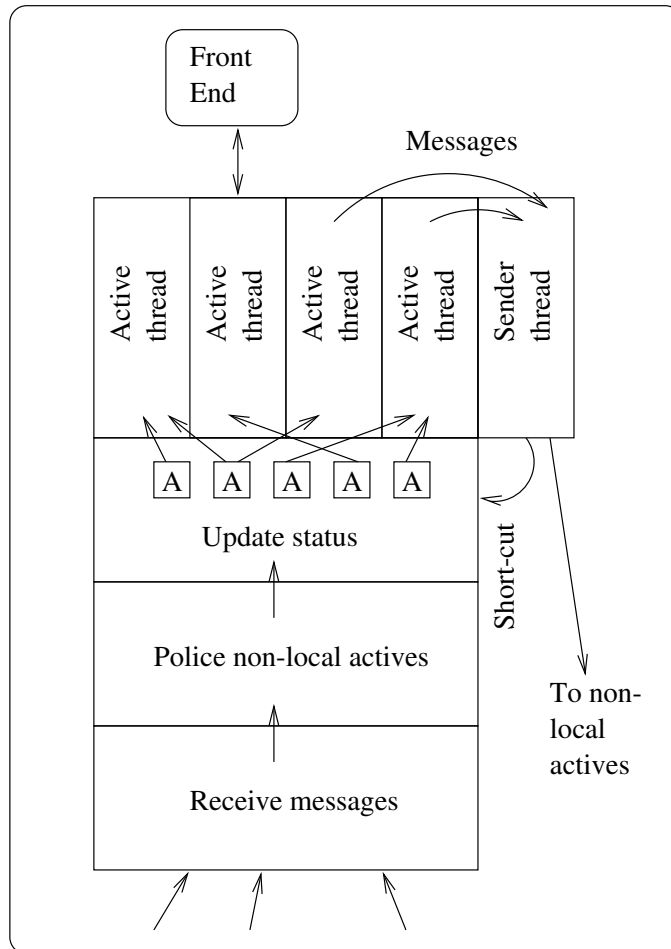


Figure 5.1: The block architecture of the persistent server process. Notice that to ease implementation of the actives themselves, the design calls for heavy use of threads. `A` is an `Active` object. Although we do not show it in this figure, it is possible for there to be multiple front-end processes, although there will only ever be at most one front-end connected to each active thread. Front-ends do not necessarily need to be graphical, and could in the case of very specialized actives such as trading posts be merely command line tools to allow the owner to inspect and configure the active.

```

        ActiveThread myThread,
        ActiveInfo myInfo)
{
    // Initialization
    ...

    // Main loop
    for (;;)
    {
        // Work out whether or not to fire an
        // engine, fire a laser, rotate, protect,
        // and so on, and if so send out a message
        // stating my intention.
        ...

        // Sleep for a short time.
        sleep (200ms);
    }
}

```

As you can see, the active thread runs in real time and responds to external events as they happen. To do so, the active may consult (read only) the database of `Active` objects stored and managed by the status thread. In figure 5.1 these objects are represented by `A`s. Each `Active` object represents the most current knowledge about each active in the sphere of influence. Active threads can query the status thread to obtain a list of all `Active` objects in their own sphere of influence. They are also passed a handle to their own `Active` object as an argument (`myActive`). They may also be passed other handles in the general “informational” structure handed to them as another argument (`myInfo`). This argument originates, for missiles and the like, from the space ship that launches them and may contain a list of targets which they must hunt and destroy. One must note when programming the smarts into actives that the information in the `Active` objects is not necessarily current, even for other local actives (although it is very likely to be current or almost current for local actives). Active threads move by calling the sender thread and passing details of their intention to move. At the lowest level, the sender thread offers methods which basically correspond to the primitive status messages like `ACCEL`, `ROT`, `PROT` and so on. However, the active threads have access to library functions which allow more complex manoeuvres to be carried out, and remove some of the more burdensome tasks such as calculating engine burn times from the programmer.

The sender thread queues primitive status messages from the active threads for delivery to their respective multicast groups. It eventually delivers messages to non-local actives using whatever reliable multicast protocol is in force, and

short cuts messages to local actives directly through to the status thread<sup>1</sup>.

Messages sent by non-local actives are received at some point by a receive thread and delivered directly to the policing thread (which does what you would imagine). After policing the messages, they are delivered to the status thread which updates the **Active** objects with the newly acquired information about their positions. In practice the tasks of policing and updating are somewhat more intertwined than figure 5.1 shows, and this is explained below.

## 5.1 Active threads

As described above, active threads provide a place to run the smarts controlling the active, in some cases directly and autonomously, in other cases proxying requests from a GUI or other front-end. For each active thread running in a persistent server, there is a single instance of the `uk.co.demon.annexia.fleet.ActiveThread` class, derived from `java.lang.Thread`.

### 5.1.1 Associated Active object

Every active thread has an associated **Active** object, which records its current status. Active threads send out messages, these messages short-cut round to the status thread, and the status thread makes the appropriate updates to the **Active** object, just as if the active were running remotely.

This implies some rather strange things about the architecture of Fleet. For example, suppose the smarts in an active thread order an engine burn for 0.5s. The code simply supplies a list of components to burn and assuming that the engine has sufficient throughput (and various other policing requirements) the code needs to know nothing more. It does not need to calculate how fast it will eventually be going, or how its speed might be affected by the burn. What happens instead is that the status thread, at some point in the future, takes into account the engine burn and updates the **Active** object. **Active** objects existing in other persistent servers and relating to this active thread are all also updated in the same way.

In reality, of course, it is often not very useful for the active thread to just order an engine burn. Instead, its requirement is that it accelerates to, say, 20 m/s. To do this, it calls a library method in **ActiveThread** which *estimates* inexactly the amount of fuel required to achieve the speed by reversing calculations done in the status thread. The estimate is based on the current **Active** state which may be out of date, not accounting for other engine burns which have already taken place. The actual speed achieved may be 18 or 22 m/s. So the best way to code active threads is to ensure that they iteratively

---

<sup>1</sup>Recall that since all local actives belong to the same player, no policing needs to be performed on local-local transactions.

make small adjustments to their speed and direction rather than attempting complex manoeuvres and speed changes with a single engine burn.

### 5.1.2 Creation

The persistent server starts by running a single `ActiveThread` object which corresponds to the player's spaceship assembly which has been serialized into a file from a previous session. Initially, this active thread has no sphere of influence (except for itself) and contacts a drop-in controller to introduce itself to the game and gain an initial sphere of influence. So most of the time there is at least one active thread. If the number of active threads drops to zero (probably because the player's main spaceship has been destroyed) then the persistent server cannot continue and has to be restarted externally.

Apart from the spaceship, other active threads have to be forked from another thread. The process is as follows:

- The parent active, in its `runActive` method, creates an `Assembly` object listing the housing component, required components and cargo for the new child active. These components must be created out of whole components that the parent active is carrying as cargo.
- The parent active passes the `Assembly` object as an argument to the `ActiveThread` library function `forkNewActive`. This library function creates and sends the correct FORK message and also calls functions in the persistent server to create a new corresponding local `ActiveThread` object. This new thread does not run yet.
- If the fork is successful<sup>2</sup>, the parent active deletes the components from its cargo that it gave to the child.
- The FORK message is eventually sent by the sender thread to other remote actives in the parent's sphere of influence and the message also short-cuts round back to the status thread.
- In order to keep the status thread fairly orthogonal, FORK messages which originate both remotely and locally are treated in almost the same way. In either case, they result in the creation of a new `Active` object locally. However if the FORK message originated locally also, then the status thread will additionally locate the `ActiveThread` object and run it. This seemingly baroque way to start an active thread is required so that the active thread's initialization function can be passed a handle to its own `Active`.

---

<sup>2</sup>Is forking always successful?

### 5.1.3 Running

Active threads run as we have already seen in their `runActive` thread. This thread, running in real time, calls upon various library functions available in the `ActiveThread` class to perform various operations.

The methods available are:

*Note:* Table of methods here.

### 5.1.4 Destruction

When an active thread comes under attack and runs out of energy to drive its shields, it is immediately destroyed (Fleet does not have a concept of “damage”). Policing actives detect this situation because the active under attack fails to send out sufficient valid PROT messages to protect itself from collision, laser fire or explosion—and so failing to carry out the correct steps resulting from destruction is recognized as cheating.

The steps are:

- The `runActive` method returns.
- The active thread, noticing the return of the `runActive` method, calls the `destroyAllComponents` library function.
- This library function issues a DESTROY message which lists all components in the active’s assembly: including housing, required components, fittings and cargo. These components are also deleted locally.
- The `ActiveThread`’s `run` method exits, causing the thread to stop.

Meanwhile, the `Active` objects both local and remote corresponding to the defunct active thread note the demise of the active. Those objects continue to exist while any thread holds a handle to them, but they are dropped from all spheres of influence and other lists maintained by the persistent server, and any attempt to call any method results in an `ActiveDestroyedException`.

### 5.1.5 Self-destruction

Actives do not normally need to take steps to destroy themselves if, like missiles, they are designed to explode. In that case, their proximity to their own explosion will cause them to be destroyed. Actives can, however, also destroy themselves voluntarily by just returning from their `runActive` method.

## 5.2 Sending and receiving messages

The sender thread has three major tasks. Firstly it owns and operates a circular queue of messages sent from locally running active threads to other actives,

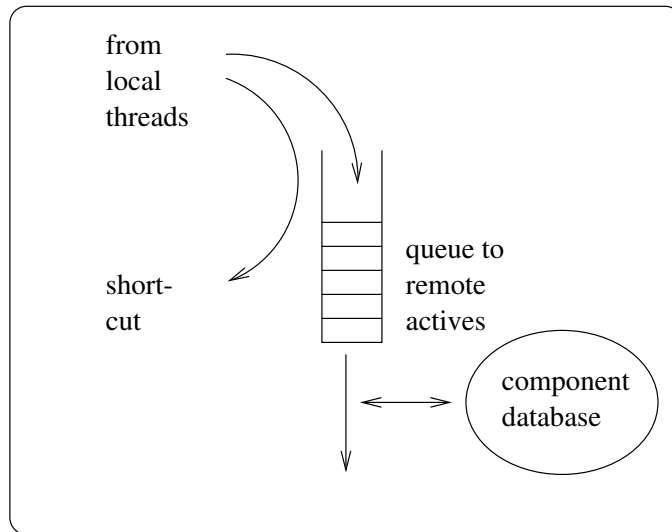


Figure 5.2: A block architecture of the sender thread.

always taking care to preserve the FIFO ordering of messages from one active to another. Secondly it determines which messages are to be delivered locally and it passes those directly to the status thread via the so-called “shortcut”. Thirdly it performs a little bit of component management, described below.

The receiver thread receives messages from non-local actives, deals with the problems of reliable delivery, and eventually delivers those messages to the status thread for policing and to update the status of the local `Active` objects. It also performs a modicum of component management.

### 5.2.1 Sender thread

The sender thread is architected as shown in figure 5.2. In concept, the sender thread is quite simple. Active threads running locally make lowlevel calls to the sender, these calls corresponding almost directly to the various types of status messages that can be sent, such as `ACCEL`, `ROT`, `PROT` and `FORK`. Since all messages are implicitly sent to all other actives in the sphere of influence, and since the sphere of influence always includes the sending active, the sender immediately invokes the short-cut route and directs the message to the status thread. If the sphere of influence also contains actives that are not local, as will usually be the case, the sender also appends the message to a queue of outgoing messages<sup>3</sup>.

---

<sup>3</sup>Note that this all happens in the context of the active thread, using synchronized methods present in the sender thread.

Messages stored on the outgoing queue are stored in a format which is internal to the sender thread, not in the format used on the wire for messages. The reason will become clear in a moment.

Periodically the sender thread wakes up on a synchronization signal and inspects the queue for messages to send. It picks the oldest message and constructs a network status message in the format used “on the wire”. To do this, it has to associate **Component** objects contained in the message with fully wrapped (signed) components contained in the **component database**. Depending on the message type, it may also wrap the components it finds in **Destroyed-Component** objects before sending them. It may also delete those components from the component database.

### 5.2.2 Receiver thread

The receiver thread is woken up by incoming messages and runs the reliable multicast protocol to read the messages.

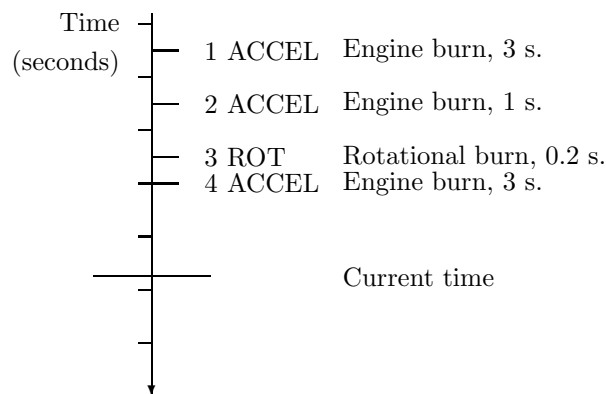
The receiver thread is very much simpler than the sender thread. It delivers up the message, essentially unmodified to the status thread which polices it.

## 5.3 Active objects

Before discussing the important part of the design responsible for policing and updating the active objects themselves, we look in detail at the **Active** class.

An **Active** object has to somehow contain the current status of the active to which it corresponds. However, actives are usually constantly moving, and the **Active** object does not necessarily have a complete picture of all messages sent by the active (although because the multicast mechanism is FIFO reliable, it does at least know about all messages received up to a point in time).

The **Active** object therefore has a picture of the state of the active that looks like this:



In the picture, no messages have been received between message number 4 and the current time, but because of network latency it is possible that further messages will be received. Therefore the current position returned by the `Active` object is always based on estimation. The default algorithm for this estimation is dead reckoning (which simply assumes that no more messages will be received between the last message and the current time).

The `Active` object stores a history of messages sent by an active, and this history is kept for a certain number of seconds which can be calculated by knowing how often actives are required to send something (or an empty but timestamped heartbeat message). Policing can be delayed by up to as long as this period and so the history must be at least this long. The `Active` object contains methods which can calculate the active's position at any point during this period of time, with points before the latest message being of definite historical record, and points after the latest message being estimates based on dead reckoning.

## 5.4 Policing and updating status

*Note:* When the SOI protocol has been finalized there will be other basic message types which must be added here.

Not all messages can be policed in full as soon as they are received. For this reason we split policing into two steps: policing done immediately as and when each message is received, and policing done later.

### 5.4.1 Immediate policing

The steps taken to police messages immediately the message is received are shown in figures 5.3, 5.4, 5.5, 5.6, and 5.7. The policing steps are described in more detail in the following sections.

#### Well-formed

On receipt, all messages are checked to ensure that they are well-formed. Fleet messages are just serialized Java objects, so we use the ordinary precautions to ensure that the objects correspond to classes which exist, with the right version, and derived from the `NetworkMessage` class. All messages must also be signed by the sender, and we check this signature here too.

#### Timestamp

All messages must contain a timestamp, indicating when the message was issued, in real time (UTC). We perform various checks on this timestamp field here, including ensuring that the active has sent out messages sufficiently often, that the timestamp does not go backwards, and that the latency implied by

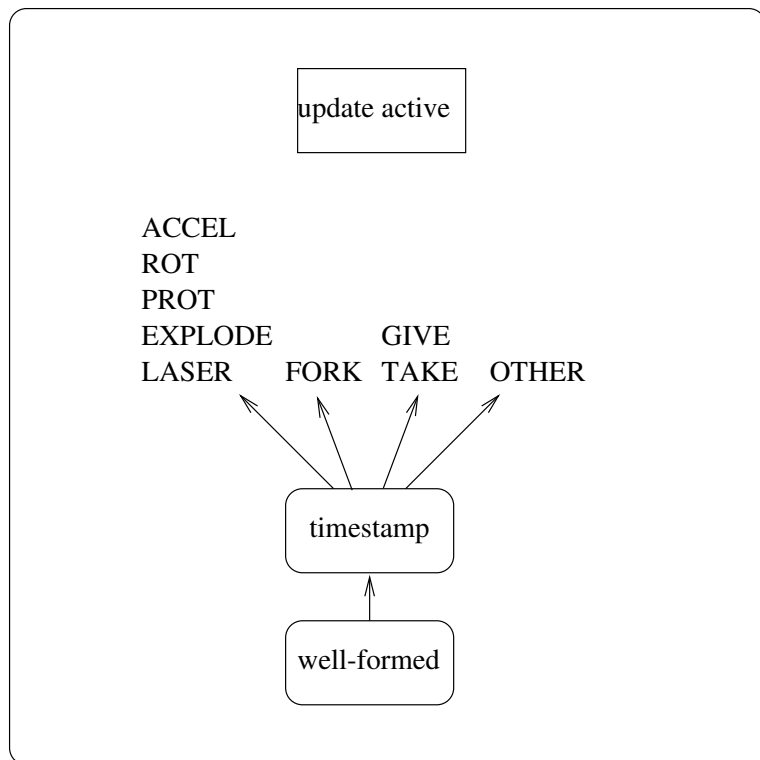


Figure 5.3: The steps taken to police messages of different types as they are received.

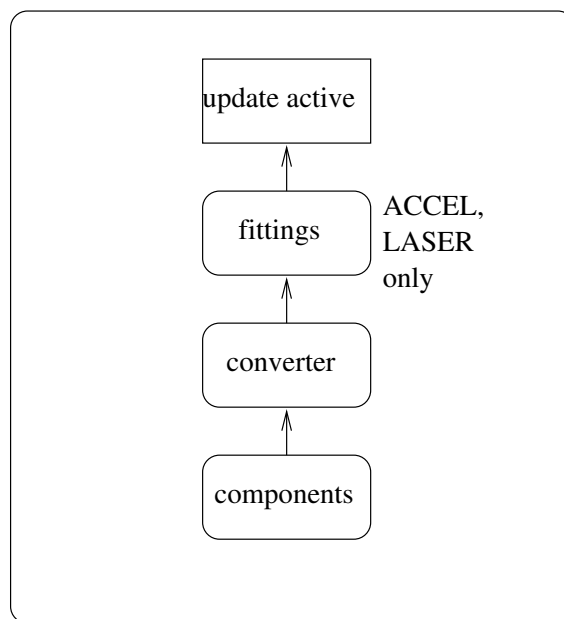


Figure 5.4: The steps taken to police ACCEL, ROT, PROT, EXPLODE and LASER messages as they are received.

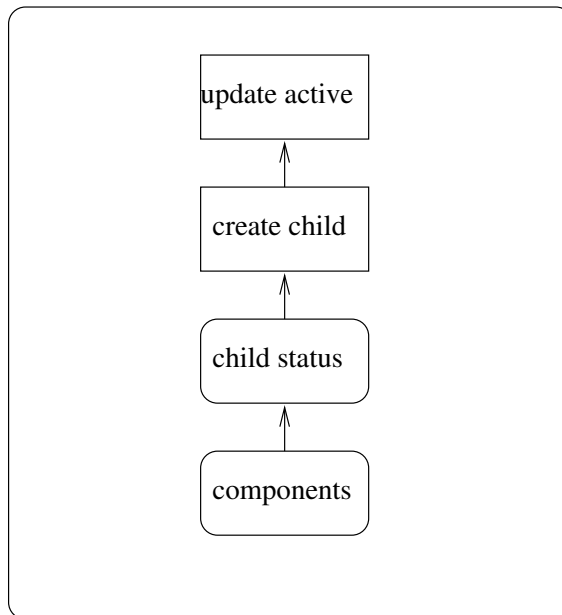


Figure 5.5: The steps taken to police FORK messages as they are received.

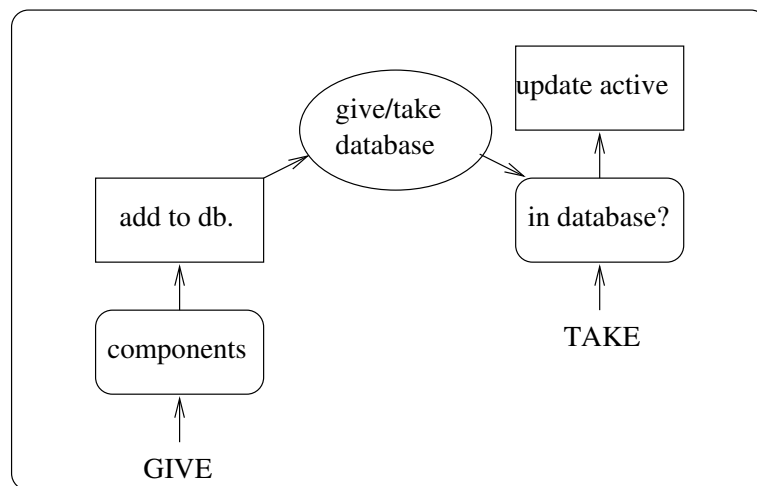


Figure 5.6: The steps taken to police GIVE and TAKE messages as they are received. Note that GIVE and TAKE messages are matched up through a common unique identifier present in the messages. The database expires unmatched GIVE messages quite quickly.

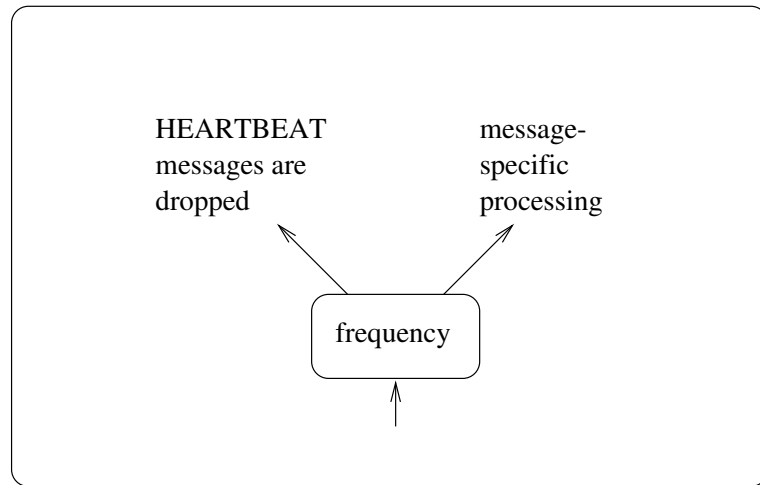


Figure 5.7: The steps taken to police other message types. Most message types are only policed for frequency of message generation.

the timestamp does not slew quickly one way or another or go outside a fixed permissible range.

### Components

Certain types of messages contain components either destroyed or transferred from one active to another. In that case, we police these components to ensure correct ownership and correct signatures in the chain of ownership. We check that the components belong to a correct class and that their issuer is correct for that class. We also check that the components originally belonged to the active sending them out. This part of the policing involves consulting a list of components known to be held by the active and, for destroyed components, passing them on to a destruction post.

### Converter

Certain types of messages contain converters which are checked by running the conversion code to ensure that the conversion was carried out correctly. We also check that the throughput of the converter is not exceeded by reference to previous manoeuvres made by the active.

### Fittings

ACCEL and LASER messages refer to engine and laser components which are fitted into the housing. We check that engine or laser is fitted into the housing

and that it points in the declared direction.

### **Child status**

FORK messages require that the child's status is checked. This involves checking that the child is close enough to the parent and has the same linear and angular momentum.

### **GIVE-TAKE protocol**

The GIVE-TAKE protocol is stateful, since TAKE messages depend on previous GIVE messages. GIVE messages contain a list of components which are policed in the normal way (but not actually transferred from donor to recipient). The GIVE message is instead stored in a database. When a matching TAKE message is received, the transfer between donor and recipient is completed. GIVE messages in the database expire after a fixed (short) period of time.

### **Frequency**

Non-status messages are usually policed just for frequency of message generation. This ensures that actives cannot flood the network with non-status messages. Other message-specific policing can be added here.

### **Sandboxes**

*Note:* This section will describe how sandboxes are applied to policing where we need to run code which is either downloaded from the root page or loaded from other players (via components or assemblies they supply to us).

## **5.4.2 Later policing**

As stated before, it is not unfortunately possible to completely police only when messages are received. We cannot police damage—collisions, laser hits and explosions—until “all the facts are known”, perhaps some seconds after the events actually happened.

Recall that when an active suffers some damage, it should begin to emit PROT messages until it has emitted a sufficient quantity of protection to cover the damage. It does not need to (and in general cannot) emit the protection as soon as it suffers the damage, but it must within a certain timelimit emit sufficient. If it cannot emit sufficient protection, then it must destroy itself. If an active suffers damage, but within a certain timelimit neither emits sufficient protection to cover the damage, nor destroys itself, then the active is cheating.

## 5.5 Rocket mechanics

In this section, we take some time to explore the mechanics behind rockets, and derive equations of motion.

### 5.5.1 Time segments

We assume from the work above that we can split an active's motion into fixed length segments of time. During each of these segments of time, the following conditions apply:

- The active's linear engines fire at a constant rate (the rate may be zero).
- The active's rotational engines fire at a constant rate (the rate may be zero).
- The mass of the active decreases at a constant rate (the rate of decrease may be zero, particularly in the case where no engines fire at all).

At the beginning of the segment, we know:

- The active's position ( $\vec{x}_i$ ) and linear momentum ( $\vec{\dot{x}}_i$ ).
- The active's angular position ( $\vec{\theta}_i$ ) and angular momentum ( $\vec{\dot{\theta}}_i$ ).
- The active's initial mass ( $m_i$ ).

We also know the length of the time segment ( $T$ ) and the constant burn rate ( $k$ ) of fuel per second over the time segment.

The task is to calculate functions  $\vec{x}(t)$  and  $\vec{\theta}(t)$  giving, respectively, an equation for the active's position and angular position over time ( $t$  ranges over  $[0, T]$ ). This allows us firstly to compute  $\vec{x}(T)$  and  $\vec{\theta}(T)$ , giving us the starting point for the segment after this one. It secondly allows us to compute the active's position at intermediate times throughout the segment, allowing us to check for collisions and laser hits which happen at some point between the start and end of the segment.

### 5.5.2 Rockets

In Fleet, all engines are rockets and are modelled as rockets<sup>4</sup>. A rocket works by ejecting particles of fuel in the opposite direction to the motion desired.

Rockets have two main characteristics: the mass of fuel that they can eject each second, and the exhaust velocity—the speed at which fuel can be ejected.

---

<sup>4</sup>The model is general enough that other types of propellantless engines can be modelled by pretending that they act like rockets, ejecting their fuel at high exhaust velocities.

Most rockets have a constant exhaust velocity, but you can vary the amount of “thrust” by varying either the mass of fuel ejected, or varying the burn time.

*Note:* The rocket equation will be explained here.

We now consider two concrete examples: the Apollo Saturn V moon rocket and the solid rocket boosters (SRBs) which lift the NASA space shuttle. Firstly, from the rocket equation thrust and exhaust velocity are linked by the following equation:

$$v_e = \frac{T}{k} \quad (5.1)$$

where  $v_e$  is the exhaust velocity,  $T$  is the thrust (a force) and  $k$  is the burn rate—which is the amount of fuel ejected each second.

The Apollo moon rocket’s first stage had a published thrust of  $3.4 \times 10^7$  newtons, and burned  $1.4 \times 10^4$  kilograms of fuel every second. This yields an exhaust velocity of around 2,500 m/s.

The space shuttle SRBs have a published thrust at sea-level of  $3.3 \times 10^6$  pounds ( $1.5 \times 10^7$  newtons). The burn rate is somewhat harder to establish, but from what we can find out, the SRBs burn in two phases, lasting approximately 50 seconds and 70 seconds respectively. In the first phase the rockets deliver 100% of their possible power and in the second phase the rockets throttle back to 66% of full power output<sup>5</sup>. If we assume from this that the SRB burns about 52% of its total  $5.0 \times 10^5$  kg fuel in the first phase of flight, then we calculate an exhaust velocity of around 2,900 m/s.

Notice, perhaps surprisingly, that in the 20 years of development between Apollo and the space shuttle, the exhaust velocity has not been increased by very much. In fact, 3,000 m/s is reckoned to be about the limiting exhaust velocity for current chemical engines.

Other technologies, notably ion engines, achieve much greater exhaust velocities. However these can eject only small amounts of mass, and so the effective thrust is small.

### 5.5.3 Gravity

The equations of motion below assume that there is no gravity. This is a good approximation, but in the long term we would like to model gravity accurately in Fleet.

We intend to develop a model of gravity whereby the fixed bodies—planets, moons and the sun—attract actives with the normal reciprocal of distance squared force. The extra terms resulting from these fixed bodies will simply be added to the equation of linear motion ( $\vec{x}(t)$ ). The fixed bodies themselves will not move (or will move with simple elliptical trajectories).

---

<sup>5</sup>The difference in power output is achieved by shaping the propellant differently in the fore and aft sections of each SRB.

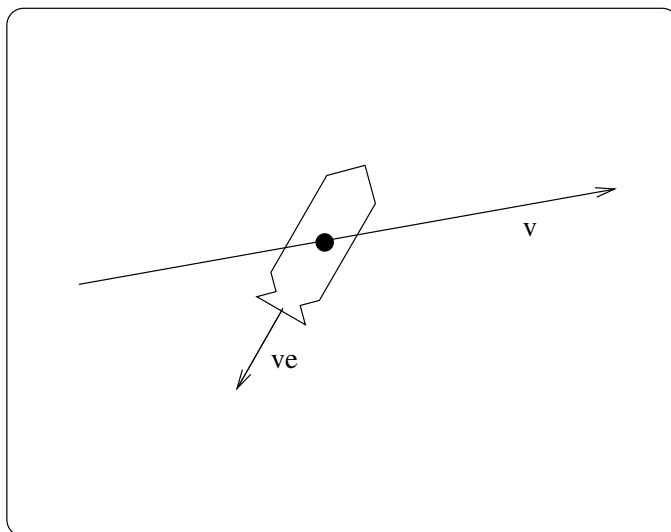


Figure 5.8: Rocket with current momentum  $\vec{v}$ , not rotating, which fires its linear engine with exhaust velocity vector  $\vec{v}_e$ .

#### 5.5.4 Linear motion

Ignoring, for now, the rotational engine, we consider first the case of the rocket with just a linear engine. The rocket is not rotating, but its linear engine does not point in the direction of motion. This situation is shown in figure 5.8. Notice that the exhaust velocity  $v_e$  has been replaced by a vector  $\vec{v}_e$ .

From the rocket equation, substituting vector quantities for scalars, we can derive an equation for acceleration:

$$\vec{\ddot{x}}(t) = \vec{\dot{v}}(t) = -\frac{\vec{v}_e}{m(t)} \frac{dm(t)}{dt} \quad (5.2)$$

If we make the simple assumption that fuel is burned at a constant rate, then we can choose a simple linear function for  $m(t)$ :

$$m(t) = m_i - kt \quad (5.3)$$

Substituting this into equation 5.2 gives:

$$\vec{\ddot{x}}(t) = \frac{k\vec{v}_e}{m_i - kt} \quad (5.4)$$

Since, in this instance,  $\vec{v}_e$  is constant this can be solved easily to give an equation for  $\vec{x}(t)$ .

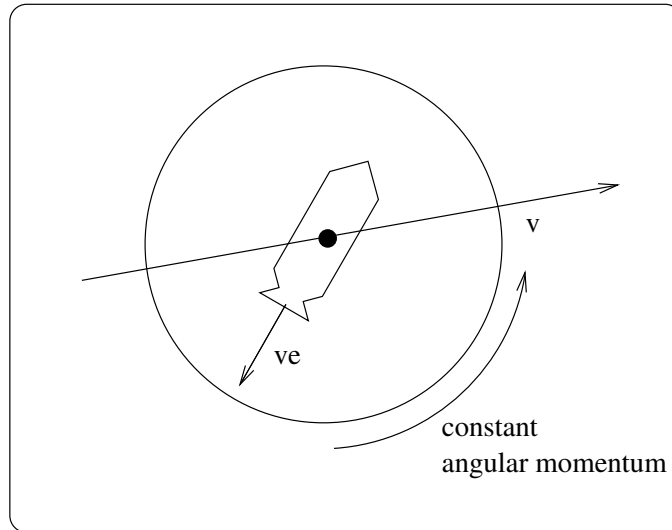


Figure 5.9: Rocket with current momentum  $\vec{v}$ , and constant angular momentum  $\vec{\theta}$ , which fires its linear engine with exhaust velocity vector  $\vec{v}_e(t)$ .

### 5.5.5 Linear motion, constant rotation

Now consider a slightly modified case where the rocket has a constant angular momentum. There is still no rotational engine firing, but now the vector  $\vec{v}_e$  is no longer a constant, but varies as a simple function of time. Figure 5.9 summarizes this situation.

Let the constant angular momentum be  $\vec{\theta}$ , and let the angle at which the linear engine points at time  $t = 0$  be  $\vec{\theta}_0$ . Then we can derive a simple equation for  $\vec{v}_e(t)$ , assuming that the exhaust gases are ejected at a constant speed  $s_e$ :

$$\vec{v}_e(t) = s_e u(\vec{\theta}_0 + \vec{\theta}t) \quad (5.5)$$

$u$  is a function which converts polar coordinates into a unit-length cartesian vector.

Substituting equation 5.5 into the basic equation for linear motion 5.4 gives:

$$\vec{\ddot{x}}(t) = \frac{ks_e u(\vec{\theta}_0 + \vec{\theta}t)}{m_i - kt} \quad (5.6)$$

### 5.5.6 Rotational engines

We can model a rotational engine as shown in figure 6.1. We assume that the rotational engine, a rocket, is attached a constant distance  $l$  away from the

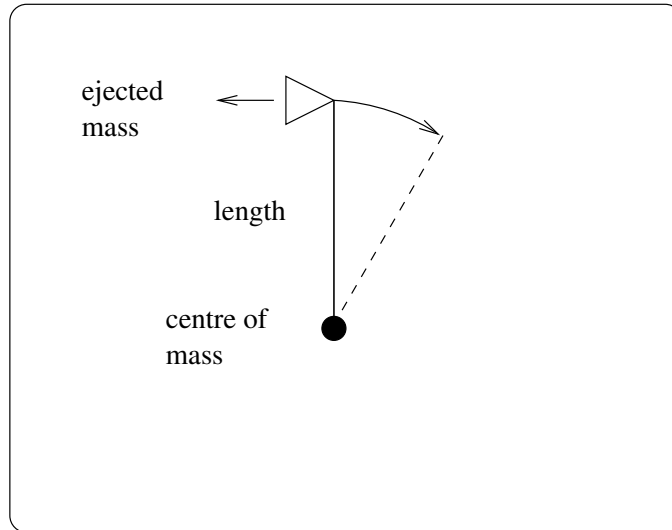


Figure 5.10: The model used for the rotational engine.

centre of mass. It ejects mass at right angles to the line from the engine to the centre of mass. This ejected mass induces a constant acceleration in angular momentum of  $\vec{\theta}$ .

*Note:* Include model and derive  $\vec{\theta}$  from  $l$  and the speed and mass of exhaust here.

### 5.5.7 Unit vector function $u$

The  $u$  function is given by:

$$u(\vec{\theta}) = M_{\theta_x} M_{\theta_y} M_{\theta_z} (1, 0, 0)^T \quad (5.7)$$

where  $\vec{\theta} = (\theta_x, \theta_y, \theta_z)^T$  and the rotation matrices  $M_{\theta_*}$  are given by:

$$M_{\theta_x} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{pmatrix} \quad (5.8)$$

$$M_{\theta_y} = \begin{pmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{pmatrix} \quad (5.9)$$

$$M_{\theta_z} = \begin{pmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.10)$$

### 5.5.8 The complete equations of motion

If the active is subject to a constant rotational acceleration of  $\vec{\theta}$ , then we can easily derive a modified equation for  $\vec{v}_e(t)$ :

$$\vec{v}_e(t) = s_e u(\vec{\theta}_0 + \vec{\theta}_0 t + \vec{\theta} t^2) \quad (5.11)$$

where:

$\vec{\theta}_0$	The direction in which the linear engine points at time $t = 0$ .
$\vec{\theta}_0$	The angular momentum of the active at time $t = 0$ .
$\vec{\theta}$	The constant angular acceleration induced by the rotational engine (see previous section).
$s_e$	The constant speed of exhaust gases coming from the linear engine.
$u$	A function which converts polar coordinates into a unit-length cartesian vector.

Substituting equation 5.11 into the basic equation for linear motion 5.4 gives:

$$\vec{x}(t) = \frac{k s_e u(\vec{\theta}_0 + \vec{\theta}_0 t + \vec{\theta} t^2)}{m_i - kt} \quad (5.12)$$

The parameter to function  $u$  in equation 5.12 can be expanded out to the following vector:

$$\vec{\theta}(t) = \vec{\theta}_0 + \vec{\theta}_0 t + \vec{\theta} t^2 = \begin{pmatrix} (\theta_0)_x + (\dot{\theta}_0)_x t + \ddot{\theta}_x t^2 \\ (\theta_0)_y + (\dot{\theta}_0)_y t + \ddot{\theta}_y t^2 \\ (\theta_0)_z + (\dot{\theta}_0)_z t + \ddot{\theta}_z t^2 \end{pmatrix} \quad (5.13)$$

Note that  $(\theta_0)_x$ ,  $(\dot{\theta}_0)_x$ ,  $\ddot{\theta}_x$  etc. are all constants.

### 5.5.9 Solutions to the equations of motion

We verified with Mathematica that there are no exact solutions for equations 5.6 or 5.12. We also attempted to find an exact solution simplifying the mass function  $m(t)$  to just a constant,  $m_i$ . In this case, the second equation has no exact solution at all, and the first equation has an exact solution with at least a thousand terms!

While there is no exact solution for  $\vec{x}(t)$ , there is a simple exact solution for  $\vec{\theta}(t)$  and that is given in equation 5.13.

Since there are no suitable approximations to  $\vec{x}(t)$  it seems sensible to construct a numerical solution from the most precise equation we have, namely equation 5.12.

# Chapter 6

## Front end

### 6.1 Front end to server network protocol

The front end process has to talk to one (or more) actives in persistent server. It is even possible, rarely, that a player would be interested in controlling actives present in more than just one server. We would like the protocol used to be efficient, perhaps using shared memory where possible. The front end and server may nevertheless be running on different machines, perhaps separated by a firewall, and thus we have to ensure that the protocol can run in full over UDP and/or TCP.

The Fleet front end protocol needs to be efficient, cross-platform, and extensible to meet the requirements of the game. These requirements dictate a simple binary protocol (efficient) in network byte order (cross-platform) with an open namespace (extensible).

The discussion that follows describes the protocol as it applies between the front end and each active. The front end may connect to as many actives as it needs to<sup>1</sup> running, if necessary, on several different persistent servers. The front end protocol is entirely private to each player and writers of custom actives are free to use their own protocols or extend the one that comes with the game. It isn't to be confused with the server to server protocol which is, of course, defined by agreement with all players and in a sense defines the game itself.

#### 6.1.1 CORBA as the underlying medium

After experimenting with a hand built protocol, I came to the conclusion that the only way to quickly and sensibly design a protocol which met the requirements was to use an existing toolkit. CORBA has the following advantages for

---

<sup>1</sup>Players then switch between different actives using a keyboard combination, rather like virtual consoles.

this purpose:

- There are a large number of free implementations for many different languages, specifically for C and Java.
- It is completely cross-platform.
- Tools are available which automatically write all the client and server stubs, deal with the underlying protocol and complicated data structures, given just a single simple input file.

CORBA has some disadvantages too, one being that it is hard to use it asynchronously, another being that language bindings aren't (yet) standardized sufficiently to allow one to port from one ORB to another easily.

### 6.1.2 Connection, naming and access control

When an active starts up which can be or needs to be controlled by the front end, the active creates a CORBA server and both prints the IOR (object reference) and, if configured to, registers itself with a CORBA naming service. The front end can take either the IOR or can resolve a name to give an IOR in order to initiate a connection.

#### Access control

To prevent unauthorized players from logging into actives which they shouldn't be able to control, actives can be configured with to use two independent access control mechanisms. The first mechanism, which is only supported where the underlying ORB supports certain low-level access to the CORBA connection, provides host based access control. The second mechanism allows the user to specify a username and password which must be presented together when logging in.

In the first case, the user supplies an access control list which lists the hosts who are permitted to connect to the server. If, for example, the front end and persistent server always run on the same workstation, then the prudent player would set the access control list for all actives to contain a single entry—`local-host`.

It should be noted that neither method is totally robust, especially where an attacker has access to the routers or machines between the front end and the persistent server, and we hope eventually to implement more robust access control based on CORBA's own security controls and strong authentication and/or encryption.

## Fleet URL

Fleet has a concept of a **URL** which can be used to refer to actives running anywhere on a local network. The Fleet URL has the following form:

```
[ fleet:// ] (IOR:... | server) [/  
[user=name [;] ] [pass=password [;] ]  
[option=(shm|udp) [;] ] ]
```

In practical terms, this means we can use the following forms:

```
fleet://IOR:.../user=name;pass=password;option=shm  
fleet://IOR:.../user=name;pass=password;option=udp  
fleet://IOR:.../user=name;pass=password  
fleet://server/user=name;pass=password;option=shm  
fleet://server/user=name;pass=password;option=udp  
fleet://server/user=name;pass=password  
fleet://IOR:.../user=name;option=shm  
fleet://IOR:.../user=name;option=udp  
fleet://IOR:.../user=name  
fleet://server/user=name;option=shm  
fleet://server/user=name;option=udp  
fleet://server/user=name  
fleet://IOR:.../option=shm  
fleet://IOR:.../option=udp  
fleet://IOR:...  
fleet://server/option=shm  
fleet://server/option=udp  
fleet://server
```

(Note that the `fleet://` prefix can be omitted).

The forms containing `IOR:...` specify the object reference directly and don't require a CORBA naming service. The forms containing `server` direct the front end to first look up the name "server" in the local CORBA naming service in order to resolve an IOR.

The forms containing `user=name` and `pass=password` direct the front end to attempt to log in to the active using the given name and/or password.

The `option` directive tells the front end to attempt to optimize communications using either shared memory or UDP-based status updates. Shared memory updates can only be used where the client and server are actually running on the same workstation (it is not possible, with CORBA, to reliably detect this situation, and so the user has to specify it themselves). UDP-based status updates are not as efficient as using shared memory but work where the front end and persistent server aren't running on the same workstation. However, UDP-based status updates won't usually work through a firewall, and in this case (omitting

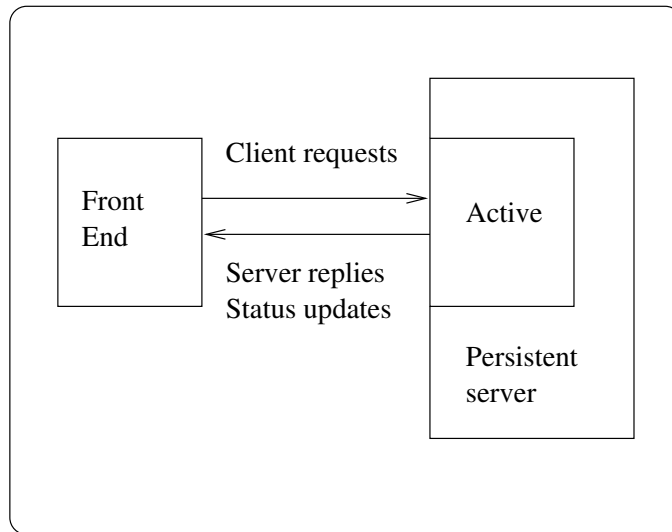


Figure 6.1: Summary of the client/server protocol used between the front-end and actives.

any option), Fleet will fall back to using CORBA for status updates, which is inefficient but should work through a properly configured firewall.

### 6.1.3 Overview of the protocol

Figure 6.1 gives a very broad overview of the protocol. There are two distinct types of message: client/server RPCs where the “client” is the front end and the “server” is the active embedded in the persistent server; and secondly status updates which are delivered asynchronously. Let us give examples of these two types of message. In the first category, we have commands such as:<sup>2</sup>

- Fire engine no. 1 for 6 seconds at maximum power.
- Trade 10g. of spices with this trading post.
- Switch on autopilot.

Status update messages, in the second category, relay information back to the front end including:

---

<sup>2</sup>At the point of writing this document, we have not decided upon a set of suitable command messages, but in the spirit of making the front end as “thin” as possible, it is likely that commands will be very high level.

- The active’s current position, momentum, Euler angles and angular momentum<sup>3</sup>.
- The amount of fuel, energy and so forth carried by the active, which are then displayed as meters.
- The positions and other information about other actives within a certain radius from the current active.

Status update messages *do not need to be reliable*. This is important. It firstly implies that some types of information cannot be carried as status updates. It secondly implies that status update messages can be carried across non-reliable connections, specifically UDP.

Because the front end has to actually draw actives that it can see, there must somehow be a way to retrieve the display information about new actives when they appear in view. This information has to be transmitted reliably, but is not time critical<sup>4</sup>. It is therefore transmitted by RPC, the front end initiating the request for the display list in response to seeing a new active appearing in a status update message.

Front ends can register to receive only particular status update information which they are interested in. They can also change various parameters including the radius of the radar scope in which they “see” other actives. This is all done using synchronous RPCs.

The next sections describe the protocol in more detail. The reader is urged to read the IDL description of the protocol, in the file `protocol.idl`.

#### 6.1.4 Login and feature negotiation

When the active starts up, it registers an instance of the `Fleet::FELogin` interface publically. Up to one front end at a time may use this interface to login and create a `Fleet::FEProtocol` transient service, through which normal commands are sent. Actives only support one-at-a-time access, and so if another front end logs in successfully, the existing `Fleet::FEProtocol` interface becomes invalid (on the assumption that the previous front end connected died unexpectedly without properly calling the `close` method).

Front ends wishing to log in first call the `login` method of `Fleet::FELogin`, passing the user name (or an empty string if no user name is required). The `login` method returns a random 64 bit seed number. The front end then concatenates the seed with the first 64 bits of the password (or 64 all-zero bits if no password is required) and passes the result through the MD5 algorithm to

---

<sup>3</sup>The front end uses a simple linear predictive method to interpolate its position and Euler angles from the time of the last status update, ensuring that (unlike Quake) status update methods do not need to be synchronized with screen updates.

<sup>4</sup>It is not time critical for two reasons: (1) actives which are within view but which do not have display information are drawn as spheres, (2) actives which come into view normally start very far away, and so would be drawn as points anyway.

form a 128 bit hash. The front end sends the seed and the hash to the `password` method<sup>5</sup>. If the login is successful, the `password` method returns a freshly created `Fleet::FEProtocol` reference, else it throws an exception.

After a successful login, the client may negotiate features. The only feature supported at present is the ability to direct status updates over either a shared memory, UDP or CORBA transport. To do this, the front end calls ones of the following methods: `enableSharedMemoryStatusUpdates`, `enableUDPStatusUpdates` or `enableCORBAStatusUpdates`. These methods throw exceptions if the relevant update method is not supported or fails for another reason.

### 6.1.5 Status updates

Status updates over shared memory

Status updates over UDP

Status updates over CORBA

### 6.1.6 Commands

---

<sup>5</sup>Note that the seed is required as well in order to associate logins and passwords, since CORBA does not normally allow us to “see” which client is making which request.

## Chapter 7

# Staged implementation plan

We plan to implement Fleet in several stages. These stages are shown in the following chart.

Stage	Date	Status	Description
1	Nov.'98	*	Draft of Fleet architecture document completed and distributed to interested parties. Comments from parties reintegrated into architecture until it becomes stable.
2	Jan.'99		Initial implementation and proof of concept for the backend. The following messages are implemented: ACCEL, ROT, PROT, EXPLODE. We implement some autonomous missiles that chase each other around and try to blow each other up. Policing is only minimally implemented. Digital signature functions are just stubs. No or only very minimal front-end. Requirement: playable between persistent servers on a LAN and as separate processes on a single machine.
3			First implementation of front-end finished. The following messages are implemented: LASER, FORK, GIVE, TAKE. We implement tools to issue components, many more different types of component. We implement spaceships, sun, planets, moons. Game released to Internet. Game still playable only on LAN and lacks complete policing and authentication.
4			Implement trading posts. Messages: PRICELIST, STARTTRADE, ACCEPTTRADE, REJECTTRADE. Corresponding changes to front-end.
5			Implement policing, authentication. Aim for full Internet-playable game.

**Part III**  
**Gameplay**

## Chapter 8

# The universe

*The year is 2230. Interplanetary space travel is affordable and convenient—if not particularly safe. The Solar System has become the new frontier. Traders vye for quick profits on the moons of the outer planets, and gangs of outlaws harry innocent travellers. The vast copper, gold and uranium concessions on Triton, the icy moon of Jupiter, attract both the honest working man and the pirate, determined to take what is not rightfully his. Iron is mined from the rings of Saturn and transported to mighty steelworks in orbit around Saturn’s moons by hired pilots—constantly at risk from attack.*

*Commander Jones picks up his flight pack and grav suit and climbs into his trusty old Phantom-V space ship. Last time he was in this baby they were both headed for the lucrative trading post around Phoebe, when they were ambushed by pirates. Wiped out, he had limped home to Earth in his broken craft. He wouldn’t make the same mistake again . . .*

*Jones flicks the ignition and the engines hum into life. “This is the Tower. You are cleared to take off. Good luck, Commander.” He turns the stick, and the ship slides round. He feels it again—the ship like a larger part of his body. This time, he’s ready to make his fortune . . .*

### 8.1 The Solar System

The Fleet universe is the Solar System, modelled as completely as possible with the Sun, all (known?) planets, all moons of those planets, large asteroids and comets. The background shows the stars as accurately as possible. The Fleet universe ends at the edge of the Solar System and available technology does not allow players to realistically explore beyond the Solar System.

The centre of the Sun is located at the origin of the Fleet coordinate system. The Earth lies on the X axis. The other planets are positioned as accurately as possible relative to the Sun and Earth, but none of the planets rotate around

the Sun (and moons do not rotate around planets). The reason is simply that Fleet does not try to model gravity—it is too complex to model and pretty much unnecessary for a good game. In effect, the Solar System is forever stuck at a particular point in time.

## Chapter 9

# Man-made landmarks

### 9.1 Drop-in points

Players can only rejoin the game at drop-in points, so there has to be a toss-up between having only a few, possibly-crowded drop-in points and having so many drop-in points that players can move unrealistically quickly around the universe.

A good compromise is to have many drop-in points located close together in only a few regions of the Solar System. We have many drop-in points around the Earth, the Moon and Mars, located near to space stations and other trading posts. No other planets have drop-in points, encouraging players to make the long trek out to distant planets through deep space.

Each drop-in point is patrolled by a large drop-in controller (see figure ??). Drop-in points are protected by peacekeepers (see section ??).

### 9.2 Space stations

## Chapter 10

# Spaceships