Version: 3/12/2003 12:58 AM

Distributed Starship

User's Guide

Distributed Starship User's Guide

FARGOS Development, LLC
757 Delano Road
Yorktown Heights, NY 10598
http://www.fargos.net
mailto:support@fargos.net

Notice of Rights

Trademarks

FARGOS/VISTA, FARGOS/SolidState and FARGOS/SolidConnection are trademarks of FARGOS Development, LLC.

Abbreviations

FARGOS Development, LLC is a Limited Liability Company registered with the State of New York. It is required to identify itself as such in its name, hence the ", LLC" suffix. For purposes of readability in this document, the ", LLC" suffix is sometimes dropped. The phrase "FARGOS Development" always denotes "FARGOS Development, LLC" and is not intended to suggest any alternate form of organization.

Notice of Liability

Information in this document is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this document, FARGOS Development, LLC shall **not** have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained within this document or by the computer software or hardware products described in it.

Contents

1. Distributed Starship

Introduction

Distributed Starship is a sample FARGOS/VISTA application that implements a space warfare simulator. It is intended to demonstrate the power and flexibility of the *FARGOS/VISTA Object Management Environment* while being entertaining at the same time. **Distributed Starship** can be played simultaneously by zero, one, or more human players. If the simulation is left unattended, it is not uncommon for it to end up pushing entire fleets of starships from one side of the galaxy to the other responding to distress messages[1]. While the entire game can be processed by a single host, **Distributed Starship** is designed to be able to dynamically split the computational workload among an arbitrary pool of heterogeneous hosts, whose membership can grow and shrink at will. In certain configurations, it can provide fault-tolerant operation.

History

The original version of **Starship** was written in 1987 by Bryan Beecher and Geoff Carpenter while they were at the University of Michigan. At the time, there was a perceived need for a game that was more realistic than the then-current genre in which a single human-controlled ship took on fleets of weak and unintelligent ships. **Starship** addressed these issues by providing a simulator that supported multiple users and a rigid enforcement of game rules to ensure that human-controlled ships had no inherent advantages over those controlled by the computer. To further enhance the realism, ships were assigned missions, such as deliver supplies to a planet or rendezvous with a ship. Completion of a mission awarded points to a ship, which could be cashed in for upgraded capabilities. In this fashion, a ship's strength increased much like hit points in a conventional role-playing game. A persistent copy of the user's ship data was maintained, thus as long as a ship survived its battles, it could be used in subsequent games.

In 1994, **Distributed Starship** was written as a demonstration of IBM Research's DRAGONS technology suite. While useful in a standalone environment, the most dramatic demonstrations were possible on a trade show floor. The original **Starship** C source code was used as a template for a new, dynamically distributed application that was written in Object-Oriented G (OOG). **Distributed Starship** was a completely new game based on many, but not all, of the concepts found in the original **Starship**—some details were dropped and new features such as a real-time graphical user interface and the ability to split the simulation across multiple hosts were added. A decade later, it was converted from OOG to Object Implementation Language 2 (OIL2) to run as a FARGOS/VISTA-based application that utilizes modern web browsers and either the new Scalable Vector Graphics XML-based application or Macromedia's Shockwave Flash as the graphic user interface. There is some loss of functionality due to the use of client/server-oriented web browsers vs. the peer-to-peer, event-driven architecture of a DRAGONS Display Manager, but the result is quite close.

[1] To keep things from getting out of hand, there is an explicit upper bound enforced in the game.

Multiplayer Synchronization

Any multiplayer game faces a significant issue related to the synchronization of the activities of the players. Many games (e.g., checkers, chess, Mille Bornes, Chutes & Ladders) have explicit rules that enforce synchronization. For example, arguably the most common rule is that only one player at a time can issue commands (e.g., make a move)—such games are classified as turn-based.

Starship was intended to operate continuously in real-time with both humans and the computer operating starships. It should be obvious that it is impossible to force all human players to enter their commands at the exact same moment in time. Likewise, computers have long been capable of performing work faster than humans, so a real-time game that accepted player input as fast as it could be provided would give a computer-controlled player a significant advantage. The **Starship** simulator addressed the synchronization issue by implementing the game as an on-going series of state transitions. Inputs would be collected from both human players and the decisions that were made on behalf of the computer-controlled starships. The collected inputs would be applied to the various objects being processed by the simulation (e.g., starships, starbases, and planets) to yield a new state for the simulation. The issue of the disparity of computational speed between human and computer was handled by imposing a delay of several seconds between each state transition: this created a window of opportunity for human players to enter commands. The disparity of processing capabilities between different humans was addressed by imposing an upper bound on the number of inputs that would be permitted by any player between state transitions (for example, a player might be limited to a maximum of three commands).

These concepts were carried over into the implementation of **Distributed Starship**:

- Rather than run flat out and compute state transitions as quickly as possible, the simulation pauses for several seconds after every turn.
- User commands are collected for application as input to the next state transition of the simulation, rather than being applied immediately. Thus, when a weapon is fired, the damage is not computed at that instant in time. Instead, the user's request to fire the weapon is queued for processing. This also permits users to enter commands at any point in time; they are always processed by the simulator on the next state transition.
- To help enforce the number of commands a user can issue per turn, some controls are temporarily disabled. For example, when a weapon is fired, the **fire** button is no longer available to be repeatedly pressed. When the weapon has recycled, the **fire** button once again becomes active.

Installing the Game

An on-going **Distributed Starship** game is maintained on www.fargos.net, so the quickest way to play is to just access the www.fargos.net site. It is, however, a public server and individuals may wish to setup their own private games. The FARGOS/VISTA Software Development Kit includes the OIL2 Architecture Neutral Format object code files that implement all of the Starship-specific application classes and some FARGOS/VISTA Object Management Environment *rc* files. A minimum of one server must be deployed for each **Distributed Starship** game, which is referred to as the *master*. Additional servers may participate in the simulation and they are referred to as *slaves*. The discussion of exploiting multiple hosts is deferred until later (see the section "*Using Multiple Servers*"). Readers, who have no desire to operate their own games or contribute a slave processor to the

public game during their participation, can skip ahead to the section titled "*Joining a Distributed Starship Game*".

The graphical user interface is provided via a World Wide Web browser, so an **HTTPdaemon** and affiliated objects must be configured. The *FARGOS/VISTA HTTP Server Programmer's Guide* covers this topic in full in and definitively, but a prototype configuration is illustrated below in Figure 1:

Figure 1

```
PeerRegistry
AnnounceServices
HTTPcommonLogFormat /tmp/http.log www.domain.com
HTTPdaemon http.profile tcp:0.0.0.0:4321
HTTPpurgeCache 30 www.domain.com
```

Note: administrators may find that the creation of an **AnnounceServices** object is not performed in the *rc* file of many existing **HTTPdaemons**. An **AnnounceServices** object is required to enable the distribution of the simulation across multiple hosts. If no use of this capability will be made, then the **AnnounceServices** object can be omitted; however, since **AnnounceServices** consumes no CPU cycles unless inter-FARGOS/VISTA Object Management Environment connections are established, there is little incentive to not enable this service.

Security Note: the presence of an **AnnounceServices** object does not enable incoming FARGOS/VISTA peer connections, so the presence of this object does not introduce a security concern. The class that is of concern is **AcceptPeerConnections**.

The *rc* file shown below in Figure 2 illustrates the objects that must be created to configure the master server for a **Distributed Starship** game. The various **LoadOIL2File** objects load the OIL2 Architecture Neutral Format (OIL2 ANF) object code files. Once the OIL2 ANF files are loaded, the initial objects needed to setup[2] the game may be created. The **StarshipHTTPsetup** object takes one mandatory argument, which is the logical name of the **URLdirectory** corresponding to the HTTP server to which the game's graphic user interface providers should attach. In both Figure 1 and Figure 2, this has been denoted as **www.domain.com**. The appropriate name must be substituted when creating a local *rc* file. It can also be provided an optional Boolean flag which indicates whether or not Shockwave Flash should be used as a rendering engine rather than Scalable Vector Graphics (SVG).

The last object created in Figure 2 is **StarshipMaster**. For a given **Distributed Starship** game, there is only one master server, which is the server upon which the **StarshipMaster** object was created. There can be an arbitrary number of additional servers participating in simulation, each of which is viewed as a slave server. The slave servers are configured identically to the master server with one exception: rather than create a **StarshipMaster** object, a **StarshipSlave** object is created instead.

[2] As an alternative, an **AutomaticClassLoader** object can be used, which will automatically load any needed classes on demand.

Figure 2

```
LoadOIL2File "file:clSS_Master.o2o"
LoadOIL2File "file:clSS_ShipNames.o2o"
LoadOIL2File "file:clSS_Missions.o2o"
LoadOIL2File "file:clSS_HTTP.o2o"
LoadOIL2File "file:clSS_HTTPsvg.o2o"
# Setup HTTP services
StarshipHTTPsetup www.domain.com
# create StarshipMaster only on master machine
StarshipMaster
# StarshipSlave is automatically created by StarshipMaster
```

There are three additional template files that must be deployed:

Table 1

File	Notes
starshipLogin.html	A server-side-include-processed (see **HTTP_SSIprocessor**) template file that must be installed in a file system directory that is exported as part of the logical root of the HTML page tree.
starshipGUI.html	A template for dynamically generated pages that display each starship's status.
starshipGUIswf.html	An equivalent template used when Shockwave Flash is the rendering engine rather than SVG.
animAlert.svg	A template for a Scalable Vector Graphics file that implements the animated alert status graphic. This file is read by the **StarshipHTTPsetup** object.
anim[Red|Green|Yellow]Alert.swf	Shockwave Flash animations for the red/green/yellow alert status graphic.

2. Playing Distributed Starship

Joining a Distributed Starship Game

The user interface for the FARGOS/VISTA-based version of **Distributed Starship** game is provided through a World Wide Web browser. A player's web browser must support the rendering engine chosen by the administrator, which is either Scalable Vector Graphics or Macromedia's Shockwave Flash format.

To join a game, a user must first browse the **starshipLogin.html** page provided by either the master server or any one of the slave hosts that are providing the HTTP-based user interface services. If only one server is hosting the game, the choice of web host is trivial. However, in the more interesting case in which the computational workload of the simulation is shared among multiple servers, a user should attempt to connect to the server that is closest to his machine. The optimal case is when the closest **StarshipSlave** object is actually resident on the user's own computer since this means she has contributed the CPU resources of her machine to assist in the

processing of the game. The use of multiple servers is discussed later in the section entitled "*Using Multiple Servers*".

Figure 3

As illustrated in the image of Figure 3, the **starshipLogin.html** page provides three options:

- Create a new starship and take command
- Scan an existing starship (which provides a read-only view)
- Take command of an existing starship

Players will normally utilize the first option, *Create a New Starship*, to create a new starship of their very own. A user may provide a desired name for the starship or let the game choose an appropriate one automatically. If a player provides a name, it should not have spaces. A player may also choose the side to join by selecting the desired race.

Visitors can monitor the status of both player ships and computer-controlled craft by utilizing the second option, *Scan a Starship*. The list of known starships is automatically generated using a server-side-include directive; if the data is obsolete, a simple refresh of the page will yield a current list.

If a player has already created a starship, but left the game and wishes to resume playing, the third option, *Take Command of an Existing Starship*, is available. The game will not permit a user to take command of a starship that already has a currently active captain.

Regardless of the option taken, a successful request will yield a display similar to the display of Figure 4 below. The fundamental difference between a read-only view and a controlling display is the presence of additional form fields, which permit the alteration of various parameters, such as a ships destination and speed. The layout of the displays and the operation of the controls are discussed in the next section.

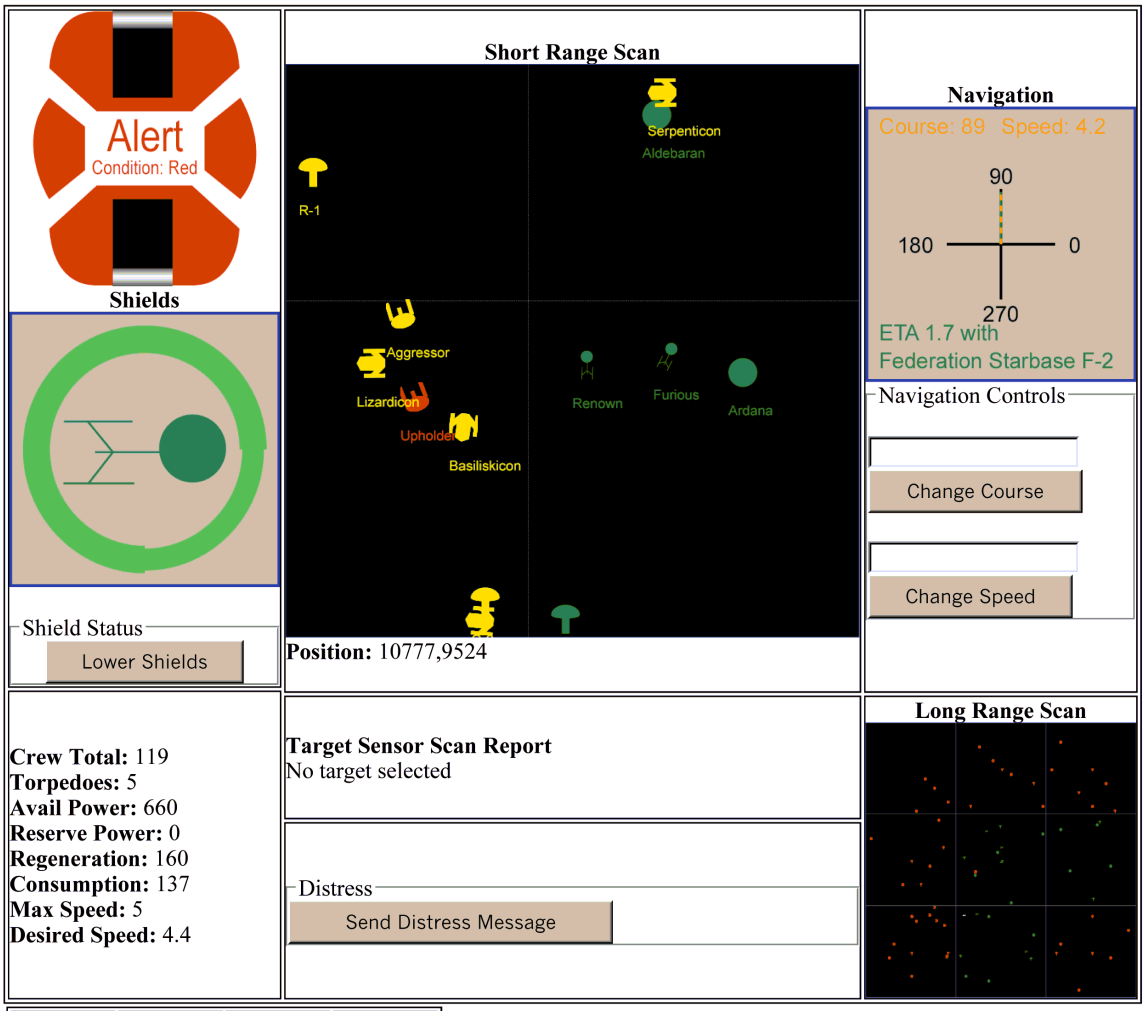
Explanation of the Display

The bridge displays provide information about:

- Deflector shield status, (and controls to raise/lower the shields).
- A short-range sensor scan and target information window.
- A long-range sensor scan.
- A display of course/speed/destination (and controls to set these to desired values).
- Weapons status (and associated controls to lock onto a target and fire).
- Miscellaneous statistics such as total crew members and power utilization.
- Text windows that display informative messages and current mission orders.

A sample display appears below in Figure 4 and Figure 5. As enhancements may have been made to the game since the point in time when this documentation was generated, the actual appearance may be slightly different with respect to some cosmetic details.

Figure 4



The Renown is under attack by the Upholder

Alert Condition

The current alert condition (green, yellow or red) is displayed using an active graphic in the upper left-hand corner. In Figure 4, the starship is under attack, so its alert condition is set to red.

Shield Status

In the upper left hand corner of the display, immediately below the Alert Condition graphic, is the current shield status. Shields can be raised or lowered by pressing the Raise Shields or Lower Shields button, respectively. The icon of the ship is surrounded by an image indicating shield strength: the thicker the line, the greater the shield strength. Shield strength is also indicated by color: green, yellow and red indicate decreasing shield strengths.

Strategy note: The shield facing an enemy will take the brunt of an attack. Rotating the ship to present a stronger shield will provide more protection.

Status Window

Miscellaneous statistics such as power usages, a list of attacking ships, etc. will appear in the status window immediately below the Shield Status display. Certain parameters will be colored yellow or red to warn of impending problems.

Short-Range Scan

The short-range scan is located in the upper center of the screen and takes up approximately 50% of the available screen real estate. The short-range scan shows ships, planets, and starbases in the immediate area. The long-range scan is used to view the location of all objects. Friendly ships will be colored in green, potential enemies are in yellow and starships actively attacking your ship will be in red. Borders between quadrants will be displayed as dashed gray lines.

All of the displayed objects are selectable by the mouse. If any displayed object is clicked upon, it will be set as the current target. Information about the selected target will be made available in the Target Display. The current target is used by all of the Weapons Controls to lock weapons and it serves as a default destination for making a course change request using the Navigation Controls.

Target Display

The target display is immediately below the Short-Range Scan. Information pertaining to the target's distance from your ship, speed and heading will be displayed. The current target is used by all of the Weapons Controls to lock weapons and it serves as a default destination for making a course change request using the Navigation Controls.

Distress Button

The distress button is located below the Target Display, which is found just below the Short-Range Scan. If a starship is under attack, a player can press the button to send a distress message and friendly nearby ships may come to assist the embattled starship.

Navigation Controls

The navigation controls are on the upper-right hand side of the display. The ship's current course and speed are displayed in yellow at the top and the ship's desired course or destination is displayed at the bottom in green. A compass rose is displayed in the middle. The current course is drawn as a dashed yellow line and the desired course is drawn as a solid green line.

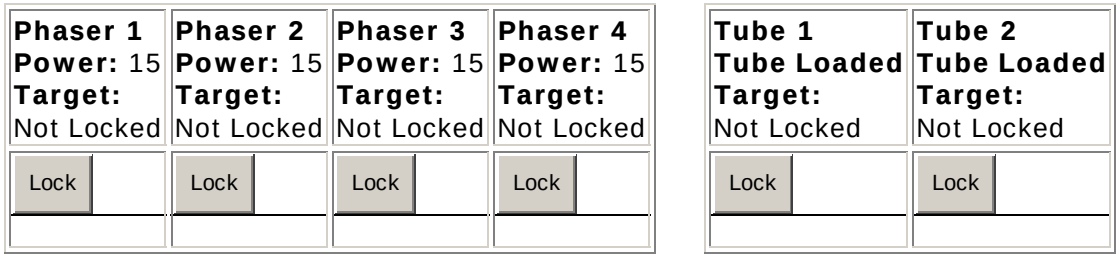
Controls to set the ship's course, speed and destination are made available below the information display. Course and destination are mutually exclusive. If a destination is selected, the autopilot will compute the required course automatically, even if the destination object moves. Destinations are specified as the short name of the object (e.g., *Vulcan* vs. *Federation Planet Vulcan*). The field will automatically be filled in with the name of the last target selected from either the Short-Range Scan or Long-Range Scan displays.

Strategy Notes: It takes a while for a starship to speed up or slow down. The faster a starship goes, the slower its rate of turn, thus one must slow down to make sharp turns. Distance traveled is a quadratic function based on speed.

Long-Range Scan

The long-range scan shows the galaxy of objects and their relative positions. It is displayed below the Navigation Controls on the right hand side of the screen. It provides functionality very similar to that of the Short-Range Scan, but it shows a much larger view. Friendly objects are in green, enemy objects are in red, and starship associated with the display will be in white. Users can also select targets by clicking on icons in the long-range scan.

Figure 5



MISSION CODENAME: Kappa Delta 4

The Federation Planet Babel was attacked by enemy forces. It is imperative that you deliver key medical supplies within 2.5 StarDays.

30 seconds Change Automatic Update Interval

Weapons Controls

The weapons controls span the display from left to right just above the Messages Window and the information about the Current Mission. These controls indicate the operational status of each weapon and they permit the weapon to be locked and fired. A weapon can be fired no more than once per turn; once fired, the corresponding **fire** button will be disabled until the weapon recycles. A weapon's target lock is taken from the target selection window. A weapon will remain locked

on a target until the target is destroyed or a new target lock is requested. It is permitted to have each weapon locked on a different target.

Current Mission

The current mission is displayed in the bottom right-hand side of the screen below the Weapons Controls. If no messages are available, the mission display may extend across the entire bottom of the screen.

Messages Window

Status messages will be displayed in the very bottom left hand side of the screen. Distress messages from other ships, messages from command, reports on an attack, warnings about having passed into enemy space, etc. will be appear here.

Update Interval

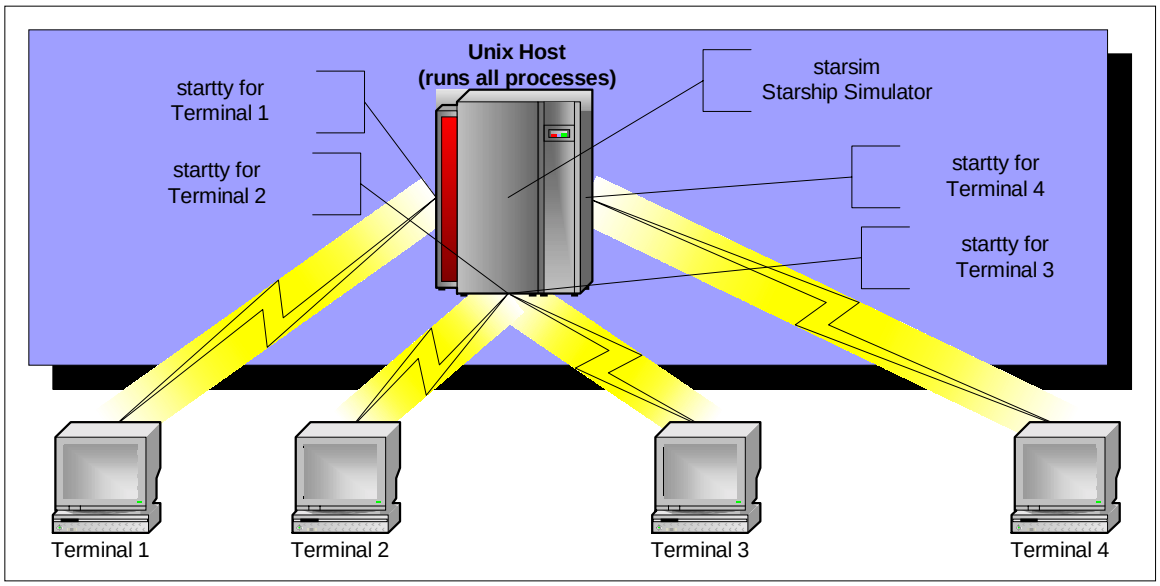
A starship's status display is normally automatically updated every 30 seconds; however, this can be set to a user-specified rate by entering the desired rate in seconds and pressing the *Change Automatic Update* Interval button, which is found at the very bottom of the page.

3. Using Multiple Servers

Implementation of the Original Starship

The original 1987 version of **Starship** was implemented using two separate applications (called **starsim** and **startty**) that executed on computers running UNIX System III or a subsequent variant (e.g., 4.3BSD, SunOS, Unicos, etc.). The single **starsim** main process implemented the simulation. A distinct **startty** user interface process was started on each player's terminal (which were typically attached via RS-232C cables). Each **startty** task accepted input typed by a player and displayed any requested information. Communication between the **starsim** simulation process and the various **startty** player interface processes took place over named pipes or sockets. This organization is graphically illustrated below in Figure 6.

Figure 6



While the communications links illustrated in Figure 6 were usually physical RS-232C cables associated with dumb terminals, the terminals could be logically realized as pseudo-ttys and displayed across a network using the **xterm** application of X10 (or the X11 of today). Regardless of how a player's terminal was realized, all of the processes associated with the original **Starship** game had to run on the same physical host.

Distribution of the Simulation

As noted above, the original version of Starship was very host-centric. Although the user interface component was broken out as a separate process from the very start, the use of named pipes created a requirement to have all processes reside on the same host. The most natural next step in distributing the workload among multiple machines was to remove the individual user interface tasks from the simulation server and place them on separate machines. This was easily achieved by altering the code to use the socket API introduced in BSD UNIX, thus yielding the capability to use TCP connections (or the equivalent). Distribution of the user interface tasks could help slightly with the scalability of the simulation; however, the original dumb TTY interface provided by the **startty** application was neither CPU nor resource intensive, thus the quantity of work that could be offloaded was not significant.

In 1994, many of the core **Starship** concepts were used in the implementation of a completely distributed, load-balanced, fault-tolerant simulator with event-driven graphic user interfaces. The current version of **Distributed Starship** is quite similar to the 1994 implementation, with the significant exception that it uses modern web browsers to provide a user interface rather than a DRAGONS Display Manager. While the use of a web browser to implement a real-time display is clearly less than optimal, it is adequate for the purposes of the game.

Since the simulator essentially performs a series of computations against each starship, starbase and planet, one obvious approach to exploit multiple physical machines would be to use a job distribution facility, such as the class **JobController**.

Another obvious distribution point is the web browser that displays each player's user interface: since a web browser does not have to run on the same host as a web

server, it is clear that the graphics processing needed to render a page can be offloaded to the user's machine. The next obvious item of processing that could be offloaded is the creation of the dynamically generated HTML pages and imagery for each user's display; however, in order to generate such dynamic content, the responsible application code needs access to all of the state information maintained by the simulator (such as a starship's speed and course, deflector shield status, etc). The short- and long-range scan displays are particularly expensive since they make a request of every single object processed by the simulator.

While the transparently distributed nature of the FARGOS/VISTA Object Management Environment would permit the processing to be handled by another host, the overhead of sending messages between physical machines would overwhelm the benefit gained by offloading computations onto other machines. Consequently, a traditional approach that uses a job scheduling application, such as **JobController**, would not be an effective means to separate the per-user graphic user interface processing from the main simulation.

Adding a Slave Server

Distributed Starship handles both problems by exploiting some of the more novel capabilities of the underlying FARGOS/VISTA Object Management Environment. When a new slave server registers itself with the master **Distributed Starship** server, a copy of each of the simulation objects is transferred to the slave. This is achieved by sending each object an **encodeObject** request. The encoded instance is then transferred into the slave system by sending the **ObjectCreator** object on the slave host an **importObject** request. As a result, all the participating systems use identical object Ids to identify information about a particular starship, but utilize their local copy whenever an application needs to retrieve information. Whenever a new starship object is created during the course of the simulation, it is propagated to each of the currently registered slaves using the same mechanism.

This organization yields several benefits. The most obvious is that it eliminates the storm of inter-host messages that would be sent by a conventional design that had everything either maintained by the master system or partitioned the objects across the processor pool of participating slaves.

When the simulation needs to process a game play step, the master instructs the slaves to perform the needed calculations on a given set of objects. Once a slave is informed about the objects it needs to process, it begins the necessary computations. All of the slaves should be performing work in parallel, although there will be some variances. Some of the factors that influence processing time are:

- The order in which they receive their respective instructions affects when computations can begin: the sooner a slave is told what to do, the sooner it has an opportunity to begin working.
- There may be differences in raw CPU speed (a 360 MHz Sun UltraSPARC IIi vs. an 700 MHz Pentium III SMP)
- There may be existing load that is unrelated to the game. With three decades of support for multi-user operation under its belt, UNIX servers often have more than one thing to do at a time.

Whenever a slave server updates a simulation object, it sends a copy of the changes that it made to the master. The master will in turn distribute the updates to the other slaves. By sending only the changes, two obvious benefits are obtained:

1. The amount of data sent is minimized.

2. It is possible for parallel work to be performed on an object as long as distinct attributes are modified. For example, the course and speed of a starship can be computed at the same time firing of a weapon is processed.

Note: if the transmission of the updates could be sent using multicast, it would be more optimal if the slave could forward the updates directly to all of its peers in a single operation. Unfortunately, in practice, this is not feasible. Sending the updates to the master might appear to introduce additional overhead: a hop to the master and then onto a slave peer. It does not. The slave originating the update would have to send the update message to the master regardless. Since the responsibility for transmitting the updates to all of the remaining slave peers resides at that point with the master and the slave sends no further messages, no additional overhead is incurred.

Loss of a Slave Server

While the design of an application that can tolerate the loss of distributed computations in progress is not an oft-practiced art, it was a guiding principle behind the design of the **Distributed Starship** classes. The 1994 version of **Distributed Starship** was intended for use at technically oriented trade shows (such as InterOp) where all of the demonstration machines were networked together[3]. Pre-configured copies of the slave server were made available for downloading from a server running in the demonstration booth. Individuals working for their respective companies could install the slave software on a machine running in their booth. Once started, it would connect back to the master machine and participate in sharing the workload. The graphical user interface component, implemented in 1994 using a DRAGONS Display Manager, was supposed to be run on a user's local machine and be connected to their local slave process, thus keeping all of the computations associated with their display off of the master server. It was possible, however, they that connected directly to the master server. The wide variety of connection options is still present in today's FARGOS/VISTA-based implementation.

One fact was known to an absolute certainty: no other company would have shipped their equipment to a trade show just to play **Distributed Starship**. Instead, their staff and computers would have been sent in order to demonstrate their own products. Thus, any slave system that was added to the processor pool could be expect to be yanked out of service at any moment when its owner decided to return it to the purpose for which the machine was brought.

Because of this operational constraint, **Distributed Starship** was designed to tolerate both the unexpected appearance and loss of any (or all) of the slave servers. Recall that all of the servers are provided with a complete duplicate of the current state of the simulation when they first register with the master. This means that the master server can accept new registrations at anytime: it does not matter how many hours the simulation has been underway; whenever a new slave is registered, it is provided a snapshot of the current state of the simulation. At the beginning of every step in the simulation, the master spreads the computational workload amongst the currently registered slaves. Since a processing step takes place every few seconds, a new slave system can be utilized in short order.

When a slave is lost during the idle period between simulation steps, it is merely deleted from the processing pool and the work to be done during the next step is

[3] It was implemented as a demonstration application to be used at trade shows by Netsmiths, Ltd., a company created by IBM Research to sell the Distributed Reliable Architecture Governing Over Networks & Systems (DRAGONS).

spread amongst the remaining (N – 1) slaves. The only interesting case comes when a slave is lost during a simulation step. Since the slave sends back updates to the master as they are computed, the effect on the simulation ranges from none (all of the updates were computed and sent prior to failure of the slave) to a complete loss of every result expected from the slave for that processing step.

There are several approaches that can be taken to recover the lost transactions. One option is to delegate the work that was not completed to another slave before moving onto the next simulation step. A more pragmatic approach is to just ignore the failure and have the work processed on the next round. It is crucial to recall that the simulation is implemented as a series of state transitions. While the slave server was lost, the original state of the objects and the corresponding inputs that were to be applied against them where not lost—they are duplicated in all of the peer systems. Thus, the unexpected failure of the slave system does not mean that a user's request to fire a photon torpedo is dropped; instead, it means that the launch of the torpedo was delayed by one turn. Although the design of **Distributed Starship** permits the system to be resilient, the approach to recovery described above is not without effect: in the meantime, starships that were processed by the surviving slave systems were able to move, thus they might now be closer or further away, their shields might have had time to strengthen, etc.

4. Applying the Concepts to other Applications

Unlike most business applications, computer games usually have significant real-time performance constraints. For example, a game that attempts to provide a frame rate of 30 frames a second will be unplayable if the frame rate drops to 4 frames a second. Even an occasional stutter will still be quite noticeable to the player. Due to its use of a web browser to support a graphical user interface, this version of **Distributed Starship** does not attempt to provide high frame rates. It does, however, demonstrate several techniques for scalability, fault-tolerance and some unique capabilities of **FARGOS/VISTA** that are directly applicable to more utilitarian applications. If one substitutes the word "*starship*" for "*independent transaction*" then many of the implementation details remain useful. Rather than plot courses, fire weapons and send out distress messages, an independent transaction could be controlling an automated tool on a factory floor, monitoring the performance of a network or system, supporting a distributed whiteboard for an online meeting, simulating a chemical process, etc.

The level of fault-tolerance supported by the current **Distributed Starship** is useful for many environments, but the master server remains the one susceptible potential point of failure. That failure point can be removed by introducing a fault-tolerant transaction monitor, such as **FARGOS/SolidState** (which provides Byzantine fault-tolerance). Such a resulting hybrid system yields the best of both worlds: near linear scalability with the survivability of a Byzantine fault-tolerant system.