M.I.T. Laboratory for Computer Science

Request for Comments No. 267
December 19, 1984

Database and Network Service Issues in, Trek, a Multi-Player Game

by Robert W. Baldwin

1. Introduction

Part of the charter for the Bell Communications Research Corp. (formerly part of Bell Labs) is to find missing links in the technology needed to utilize residential fiber-optics service. They imagine that one of the significant uses of high bandwidth residential services will be multi-player games offered by such corporations as Atari and Lucasfilms. Since I have some experience with multi-computer multi-player games, they invited me to give a talk about the technical issues that must be solved to support such games. This memo is based on that talk.

2. Overview

Trek is a multi-player multi-computer space battle game that was originally written by Eugene Ball at the University of Rochester in 1977. Each player uses an Alto minicomputer [Thacker, et al. 79] to control his or her ship and display information about the simulated universe and the other players. The Altos communicate with each other via a 3 Mbps experimental Ethernet. There are many interesting aspects to the design and implementation of Trek, but this memo focuses on communication and database issues. This memo supports two hypothesis that have been argued for by other researchers. First, that best-effort multicasting of datagrams is a desirable network service (see [Boggs 83]), and second, that application specific information can be used to simplify the design of distributed databases (see [Fischer & Michael 82]).

3. World Model of Trek

Every simulation has an underlying world model that is independent of the way it is viewed and controlled by the users. The world in Trek consists of 15 star systems that contain various numbers of stars, planets and asteroids. These are fixed objects that nether move nor change from game to game. Most fixed objects have both mass (gravity) and density, so they can influence the motion of other objects and cause damage if a collision occurs. One kind of fixed object, stargates, have neither mass nor density, instead they allow players to jump from one star system to the next.

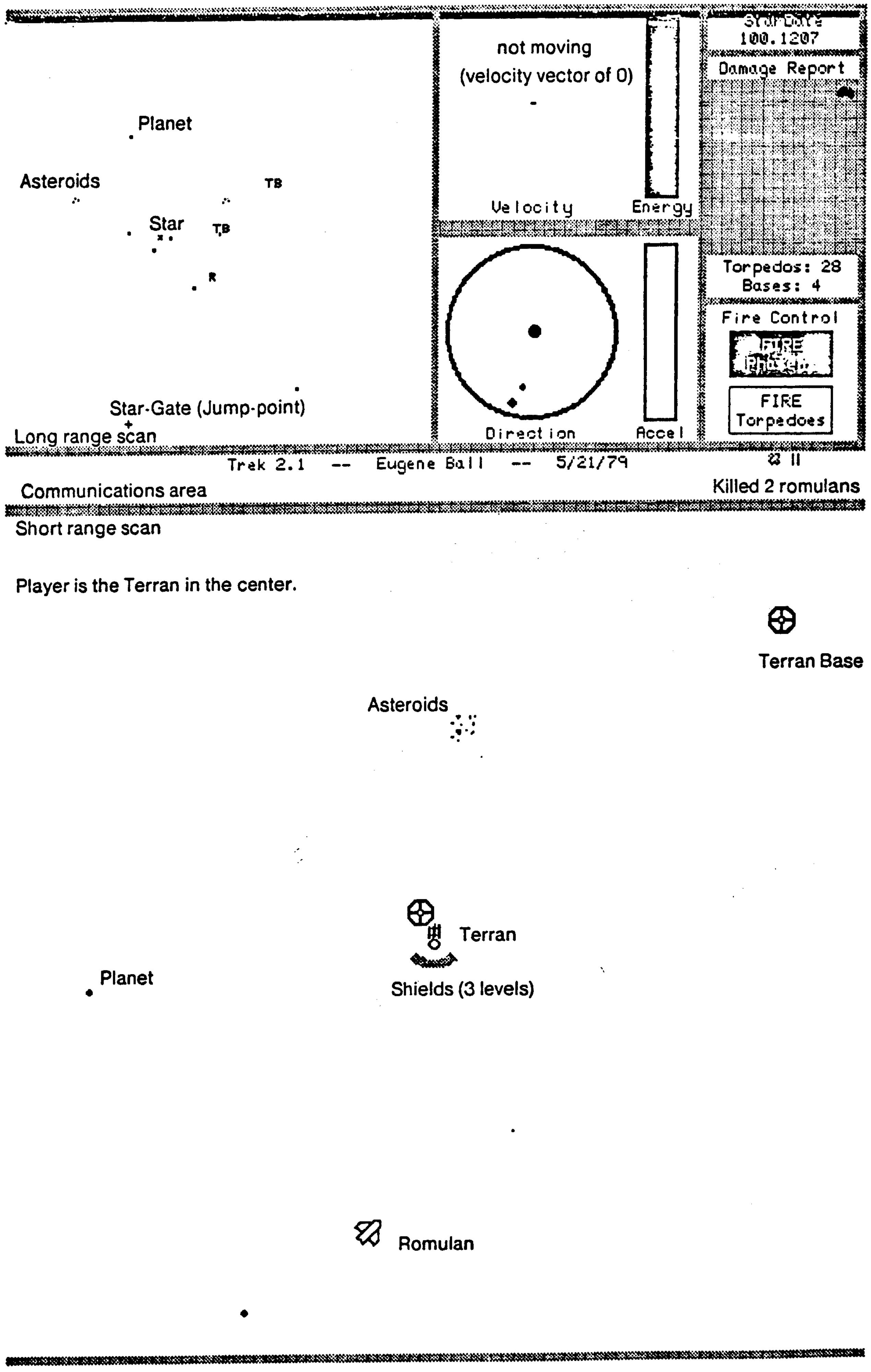
Each player controls one space ship. The ships contain two weapon systems, phasers and photon torpedoes; two engines, warp and impulse; four deflector shields and a power plant. The complexity of the ships adds an enjoyable depth to the game. To add variety to the game, there are three kinds of ships with different strong points; for example, Romulan ships turn slowly but have higher acceleration than Terran and Klingon ships.

4. User Interface

An image of the user screen is shown on the following page. The largest and bottom-most portion of the display is the short range scanner that gives detailed information about the objects surrounding a player's ship. The player is always placed at the center of the scanner as in any radar-like display. In the upper lefthand corner is the long range scanner. It displays less detailed information about objects that are far away from the player. For example, the orientation of opponent's ships is only shown on the short range scanner. I believe that working with partial information about opponents is an essential feature of competitive multi-player games.

The upper righthand corner of the screen displays the ship state and controls. The velocity of the player's ship is represented by a vector in the direction of motion with the length proportional to the speed. Energy reserves and engine acceleration are both represented as thermometers. A very important display is the damage report that list which system are not working and roughly how long it will take for them to be repaired.

Between the upper and lower portions of the display is the scoring and message area. Players use the message area to broadcast messages to each other. This communication channel is used to arrange temporary truces, to deceive players (e.g., "Help my weapons are gone") and to heap abuse on an opponent either before or after a battle.



Star Date
100.1207

Torpedos: 28
Bases: 4

```
Trek 2.1  --  Eugene Ball  --  5/21/79
```

Killed 2 romulans

Player is the Terran in the center.

5. Game Play

Typically there are up to six people playing Trek in a free-for-all style. Ships of the same kind (e.g., Terrans) do not necessarily work together. Even though any one player spends one to two minutes setting up an attack, with six players there is always a battle taking place. The battles themselves last 10 to 30 seconds depending on the speed chosen for the attack. Only about half the battles end in the death of one of the contestant. Most the time a player gets damaged and is not finished off until a second encounter. When a player does die, there is a brief pause before he or she is reborn near one of the stargates.

Notice that Trek is not a shot-em-up bang-your-dead game. Players don't die instantly and the pace is slow enough for players to communicate with each other and to plan attack strategies.

6. Network Services and Protocols

The basic strategy Trek uses to create a shared distributed database of all the players object's is to have each player broadcast the location and state of his or her objects. The basic network service used by Trek is best-effort (i.e., unreliable) multicasting of datagrams. This kind of service will be explained in the next few paragraphs. It should become clear that most multi-player games will want this kind of service rather than a reliable point to point stream.

A datagram is a collection of data with the property that either all of it is received or none of it is received. Datagrams make it easy to group together information into atomic chunks. Such grouping is an ideal service for transmitting snapshots of a player's state. An unreliable stream would make snapshots hard to implement, because the entire state may not be received.

There are two basic ways that Trek could have utilize a reliable stream as its basic transport mechanism. A reliable stream could be used to simulate datagrams by inserting markers that delimit consistent blocks of state. This approach does not have any advantages over sending datagrams and has the disadvantages of extra overhead needed to achieve reliability. Alternatively, the program could take advantage of the reliable nature of the stream by just send changes to the state rather than full state snapshots. Transmitting incremental changes to a player's state would reduce the demand for communication bandwidth, though it would increase the demand for processor bandwidth in the receiver's computer. Another difficulty with incremental state information is that all of it must be processed. If a receiver is overloaded with either I/O or the processing of incremental data, it cannot through away any of the data to lighten its load. If it did, its view of the simulated world would become inconsistent and stay inconsistent until it received a full snapshot of a player's state. Even if all the computers could keep up with the incremental state information, the programs must still have a way of sending full snap shots. Such snap shots are needed when a new player joins the game.

The reason for wanting an unreliable service is primarily that such a service uses fewer resources than a reliable one. The amount of player state information is small enough that all of it can be sent in a single datagram. New state information completely replaces the old information, so there is little benefit to insuring that old state information reaches all the players. This is just another example of end-to-end argument for reliability [Saltzer 84].

Multicasting is a term used by Xerox to describe selective broadcasting. By multicasting rather than broadcasting the game does not load down hosts that are not participating. The alternative to multicasting is sending individual packets to each of the other players. The resulting network load is proportional to the square of the number of players and thus is unacceptable for large games[1]. I believe that most multi-player games will want to use multicasting and a basic network service.

The problem with multicasting is that it is hard to implement across network boundaries. In fact Trek can only be played between Altos on the same Ethernet cable. Trek implements multicasting by sending and receiving packets to and from a fictitious, but globally known, host address. This strategy can be used because the Alto's processor can set the address used by its network interface.

I would argue that network interfaces should have several setable receive addresses. At least one setable address is necessary to implement the multicasting feature used by games like Trek. Several setable addresses allow a computer to participate in several applications that use multicasting. It seems that the trend in personal computers is toward systems that allow a user to run multiple applications at the same time with a window system to switch the users attention between them. As more applications rely communication networks to access information that is

[1]Another Alto game, MazeWars, does use individually addressed packets, but it places a limit on the maximum number of players and it has much lower bandwidth requirements than Trek.

stored or generated remotely, it will become important that the hardware not limit which applications can be run at the same time.

The major drawback of Trek's approach to multicasting is allocating a host address to every application that performs multicasting. To avoid running out of host addresses, it may be necessary to dynamically manage a table mapping application names into host addresses. If the host address space is large, then the allocations can be static. That, however, leads to the question of who allocates new addresses and how are conflicts avoided? Clearly further work is needed on this problem.

To multicast between networks it must be possible to tell a gateway to recognize and forward multicast packets. This would probably involve dynamically allocating some resource within an existing gateway or setting up a host that specifically sends the game packets to the players on other networks. Either of these strategies has the usual problems of dynamically allocating a scarce resource with the additional complication that in a distributed environment hosts can fail without warning, so it is hard to tell when a resource is no longer needed. This is an open area for further research.

An alternative to distributed multicasting is to have a high throughput central computer handle the load of rebroadcasting packets to game players. If the central computer is reliable, then the allocation and deallocation problems are simplified.

6.1. Bandwidth and Delay Requirements

The bandwidth requirement of Trek is not large from the point of view of local area networks, but it is larger than the service available with a typical modem. The driving rate for data communication is the simulation update rate. The underlying world of Trek, and the player's display, are updated twice per second. To make broadcasting more reliable, each player sends state information at least twice within the half second update period. Specifically, each player send five datagrams per second. A typical datagram contains 90 bytes including the Ethernet transport header (four bytes). Thus each player sends about 4 Kbps and with six players the total network load is 24 Kbps[2].

[2]Notice that the 24 Kbps network load is also the load on each hosts local memory, because every player receives every datagram. For fast networks, the computer's memory bandwidth will become a bottleneck before the network bandwidth does.

There is an unexpected trade-off between the transmission rate of state information and the probability that all players will receive a broadcast. The reason a packet is not received by everyone is that some computers will have temporarily run out of a resource needed to service the network device (buffers or cpu time). Increasing the number of times that a datagram is sent greatly increase the load that the network places on each computer (remember there are several computers all sending their datagrams at that rate). As the load increases, the chance of a host running out of resources and becoming deaf to the network also increases. Originally Trek sent packets ten times per second. Without doing a detailed experiment I decided to cut the rate down to five times per second. Subjectively I noticed that the performance improved and the reliability seemed the same.

One last observation about the network load is that each player contributes an amount that is independent of the number of other players. The database strategies used must preserve this property or else you get back to bandwidth requirements that increase quadratically with the number of players.

Packet delivery delay is another network characteristic that strongly influences multi-player games. Trek needs low delivery delays to ensure that each player's view of the database is close to the view seen by all other players. In general, any predictable delay can be compensated for by slowing down the game action or by decreasing the resolution of knowledge about other players. However, highly variable delay is hard to cope with.

7. Database Access and Updating

To simulate the world of Trek, the computers running Trek must cooperate to create a replicated database that contains all the fixed and movable objects (e.g. stars and ships). As objects interact with each other according to the laws of the Trek universe, the state of each object must be updated and the updates propagated to all the computers running Trek. Portions of the database are read and displayed to each player.

The two main difficulties of implementing this database are serializing access to the database (either to read or update some object) and reducing the computational load of simulating the universe. The problem of propagating changes to other computers is solved by multicasting. The general strategy used to solve the serialization problem is to assign each object to one computer and make that computer responsible for updating the object and multicasting its current state. This reduces the hard problem of having multiple readers and writers down to the simpler problem of having one writer and multiple readers. To keep the computational load reasonable, objects are assigned to computers in such a way that it is easy to filter out database activity that cannot effect the objects managed by a particular computer.

Unfortunately, the database cannot be cleanly partitioned to both solve the serialization problem and the computation problem. A typically serialization constraint is to ensure that if one player sees a torpedo hit a ship, then the owner of the ship will also see the hit. This constraint is between two objects that could be assigned to different computers. Algorithms do exist that could satisfy this constraint, but they require substantially more message traffic than the multicast scheme used by trek (see [Herlihy 84] for a description of those algorithms).

Trek resolves the conflict between serialization and partitioning by allowing one player's computer to request that an operation be performed on an object assigned to another player and by relaxing the serialization constraints. Each computer performs computations to decide how the objects assigned to it influence all other objects. When such an influence is noticed, that computer will send a request to the owner of the object being influenced. The request is appended to the normal multicasted state datagram, though it could be sent as a separate point-to-point message[3]. Part of the datagram processing includes a check for datagrams concerning objects managed by the receiving computer.

The other mechanism for resolving the conflict is to make it difficult for the people playing the game to observe a failure in the serialization of updates. This allows the use of simple algorithms that work most of the time. For example, torpedoes do not disappear when they hit a ship, they continue through it, so there is no feedback on whether a hit actually occurred. Similarly, a single hit is not enough to destroy a ship, so there is no feedback from immediate death.

The overriding criteria for assigning an object to a particular computer comes from players entering and leaving the game. When a player quits, all objects own by that player should be removed. Conversely, when a player enters the game he or she must provide the computational resources for any objects created for that player.

---

[3]The penalty for piggy-backing the request and the state information is that it has the same reliability as the state information. A request may be received multiple times or not at all.

Even with the computation divided up so that each player's computer only has to simulate the objects belonging to that player, every object (fixed or movable) can effect every one of the player's objects. For a modest minicomputer (the Alto has 16 bit words and no floating point hardware), this load is too high. Further filtering is done on the database to reduce the load. The first level filter ignores all state information about objects in star systems other than the one the player is in. Such a filter is possible because if a player in another system influences some object owned by the player (e.g., shoots the player's starbase in that system), then his or her computer will send a request that contains the necessary information. The next level of filtering is done by range. The game has a designed in maximum range at which two objects can influence each other. Lastly, types of objects that have a large number of instances, like (i.e, torpedoes and starbases), do not effect each other, so those interactions do not have to be computed. Thus, by carefully choosing the rules of the game, the shortcomings of the hardware can be avoided.

8. Conclusions

This paper has described a multi-player multi-computer game that has been popular on Alto minicomputers for several years. It is argued that the best network service for such games is a best effort multicasting of datagrams. A low delay between transmission and reception of datagrams is helpful, but not as necessary as having a predictable delay. The rules of Trek were chosen to simplify the database algorithms. A general purpose database service is neither necessary nor desirable.

References

[Boggs 83] David Boggs.
Internet Broadcasting.
PhD thesis, Stanford University, October, 1983.

[Fischer & Michael 82]
Michael J. Fischer, and Alan Michael.
Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network.
In *Symposium on Principles of Database Systems,* pages 29-31. ACM Sigact-Sigmod, Los Angeles, CA, March, 1982.

[Herlihy 84] Maurice P. Herlihy.
Replication Methods for Abstract Data Types.
PhD thesis, Massachusetts Institute Technology, May, 1984.

[Saltzer 84] J.H. Saltzer, D.P. Reed, and D.D. Clark.
End-to-End Arguments in System Design.
ACM Transactions on Computer Systems 2(4), November, 1984.

[Thacker, et al. 79]
C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs.
Alto: A Personal Computer.
Technical Report CSL-79-11, Xerox PARC, August, 1979.