IEEE Spectrum
On page(s): 32 - 38
March 1997
Volume: 34 Issue: 3

Channeling the data flood

Without careful object filtering and networking, even a thinly populated virtual world will be swamped by its own data

BERNIE ROEHLE
University of Waterloo

In many traditions, God alone is able to create a world out of nothing. True to form, mortal programmers and users of virtual worlds—more formally known as distributed virtual environments—must fall back on base material with which to ply their craft. They must rely on the distributed virtual environment (DVE) and networking software that can provide a simulated environment in which two or more participants interact with each other and with their surroundings.

In working with this software, they face great technological problems, of which the biggest are: managing the enormous data loads involved; negotiating the interplay of all the virtual objects, including the virtual people; and creating the most efficient network topologies possible for users' machines.

At the network level, each occupant of a DVE system uses a computer that is connected to a local- or wide-area network (LAN or WAN), which provides information on the changing state of the virtual environment and its contents. In DVE jargon, each of the computers in the network is called a host, and virtual objects that have any kind of changeable state are called entities or, if controlled by a human, avatars. Sometimes entities operate relatively autonomously, in which case they are commonly known as agents or bots (from "robots") [Fig. 1].

The collection of information that describes an entity in full is referred to as its dynamic state or dynamic state vector. For example, the current location and orientation of an object is part of its state. So is its color, its size, and any other property that is subject to change over time.

Data spacing, data timing

A DVE system's job is to mediate the exchange of entities' dynamic state information among hosts on the network. State changes instigated by a user on one system (such as making an avatar open a door) must be communicated to other hosts on the network. This "updating" allows hosts/users to have a consistent view of the world: when a user opens a particular door, all of the others with an unobstructed view should see it open.

Typically, a DVE system will send packets of information over a network to convey the current state of the entities in the system. When a door starts to open, for example, the system might send out a packet containing data about the door's angle of rotation around its hinge. That information would then be received on all the other systems, so that they could update the door's degree of rotation in the virtual world they are presenting to their users.

Among the key challenges in implementing a DVE system are dealing with limited network communications bandwidth, working with high and variable latency (delay in sending messages), and maintaining a virtual world consistent for all its inhabitants.

Bandwidth limitations are one of the most serious challenges. A simple approach to sharing state information would be for every entity to continually send information about its current state over the network to each of the other hosts. This setup works fine for a few entities, but, as the world grows more complex, more data must be exchanged until bandwidth becomes the limiting factor, with consequences that will become obvious later. In addition, any one client (host) computer may have to handle incoming messages that are not relevant to it, incurring an undesirable processing overhead.

Related to bandwidth is the issue of latency, the time lag that occurs as different computers receive updates about events elsewhere in the environment. When a user has his avatar walk across the room, it takes time for his instructions (sent by mouse, keyboard, or whatever) to be registered by his software. It takes more time for his machine to convert the information into a suitable form for transmission. And it takes still more time for the information to be sent over the network, especially if it's a WAN. Finally, additional time is needed at the receiving end for the data to be converted and used to update the remote host's copy of the database comprising all the entities of the virtual world.

If the total delay is too great, there is a long wait before evidence of the user's actions actually propagates to the other hosts on the network, when the user's avatar is seen to perform the action. The system thus becomes difficult or even impossible to use.

High levels of latency, as well as highly variable latency, can affect consistency. For example, if User *A* picks up a glass of water, and User *B* does the same thing at the same time, their respective machines may learn of the other's action too late. At that point, the virtual environment is no longer consistent; each user believes that he picked up the glass, and therefore that the other user did not. This suggests some amusing scenarios, but in general such inconsistencies are undesirable.

In a perfect world, bandwidth would be unlimited and latency would be negligible. Unfortunately, the Internet, despite its success and widespread adoption, is poorly suited to real-time interaction. Not only do packets of data take some time to arrive, but this latency is extremely variable and unpredictable as well.

Delays on the Internet are typically in the tens or hundreds of milliseconds (a 500-ms, or half-second, delay is quite noticeable). What's more, the overwhelming majority of the Internet's users are connected by at most 28.8-kb/s modems—truly a narrow pipe down which to send a great deal of data.

Newer communication technologies, such as asymmetric digital subscriber loop (ADSL) and cable modems, promise to increase the data rates available to the average user to the megabit-per-second range. It will be many years, however, before those systems are in widespread use. In the meantime, strategies for dealing with bandwidth limitations are critical to the success of any DVE system.

Sending behaviors

It would be easy to design a DVE system with no concern for bandwidth or latency. Each entity would transmit its entire state vector, and packets containing this set of values would be sent at an update rate sufficient to provide visual realism. In practice, current DVEs used with home modems provide a fairly jerky frame rate of four to 10 frames a second.

But in the real world, bandwidth would quickly run out, even with a modest number of users. Suppose, for instance, that a toy airplane is flying around a room under the control of a user. An update for the plane would include its current location in *x*, *y*, and *z* coordinates; each of the coordinates might be a 4-byte floating-point number, making 12 bytes in total.

The orientation of the airplane around the *x*, *y*, and *z* axes is another set of three floating-point numbers, for another 12 bytes. To distinguish the airplane from other entities, there also needs to be some kind of identification for it—adding a further 4 bytes.

That adds up to 28 bytes total per state vector. If each vector is sent 30 times per second, the total jumps to 840 bytes per second (not including any overhead imposed by the lower-level network protocols). With just half a dozen such entities in the world at once, the total bandwidth requirement is 5040 bytes per second, or roughly 50 kilobaud (10 bits sent for each byte)—more than twice the data rate of the fastest modem most people have today.

As more entities arrive in the simulated world, the need for bandwidth goes up, and it increases even faster as the complexity of the entities grows. For example, were the airplane to have movable wing surfaces, the update packets might double or triple in size, with a corresponding increase in the necessary bandwidth. Incorporating various sounds in these environments may call for even more bandwidth [but see "The sound dimension," pp. 46–50].

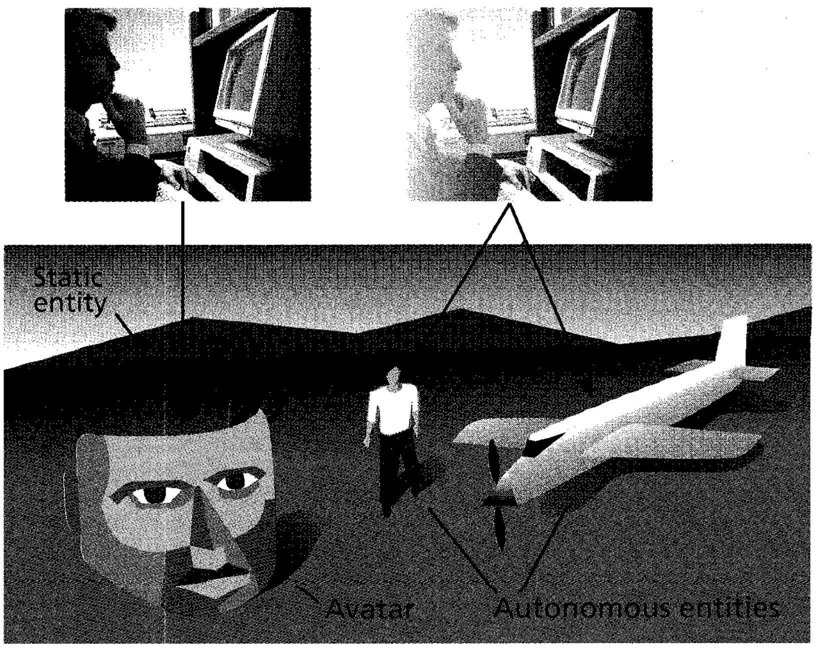
Dead reckoning

When streaming media such as audio or video are sent, bandwidth requirements can be reduced by using compression. Video compression, for example, takes advantage of the fact that one frame changes very little from its neighbor, making it possible to send only the differences between the frames.

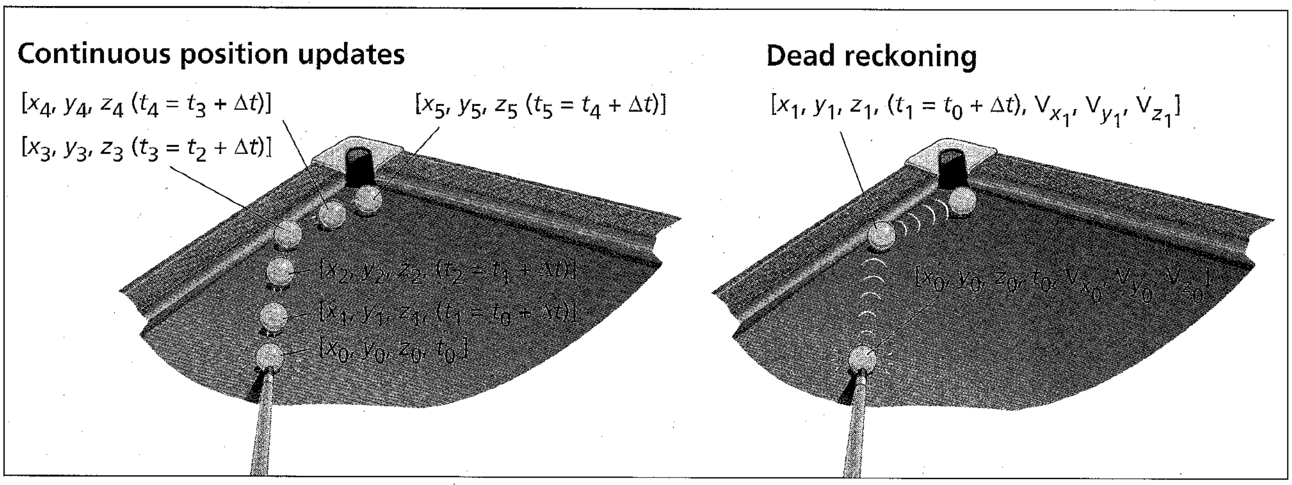
A related approach can serve to reduce the bandwidth required for DVEs. The idea is to take advantage of how similar the current state of an entity is to its previous state. Many data packets are more or less "superfluous," given the predictability of events.

Consider a virtual ball rolling across a pool table using a DVE protocol with continuous updates. The initial coordinates, speed, and direction of the ball are known. The equations of motion are stored, probably at the host [Fig. 2].

The ball is not guided by the user, but does receive an initial impetus. The equations are applied to the pool-ball entity, and the its final position is calculated. The software then calculates where the ball should appear at each



[1] Any object in a virtual world that can change its state in some way is called an entity. This scene is inhabited by the basic entity types: avatars, controlled by real people; autonomous entities, or agents, controlled to a greater or lesser extent by software alone; and static entities, which are part of the scene design.


Continuous position updates
$[x_4, y_4, z_4\ (t_4 = t_3 + \Delta t)]$
$[x_5, y_5, z_5\ (t_5 = t_4 + \Delta t)]$
$[x_3, y_3, z_3\ (t_3 = t_2 + \Delta t)]$
$[x_2, y_2, z_2, (t_2 = t_1 + \Delta t)]$
$[x_1, y_1, z_1, (t_1 = t_0 + \Delta t)]$
$[x_0, y_0, z_0, t_0]$
Dead reckoning
$[x_1, y_1, z_1, (t_1 = t_0 + \Delta t), V_{x_1}, V_{y_1}, V_{z_1}]$
$[x_0, y_0, z_0, t_0, V_{x_0}, V_{y_0}, V_{z_0}]$

[2] Different DVE techniques impose vastly different data loads. With continuous updates [left], the initial coordinates and starting time of the ball entity on a pool table are made known to all interested parties. Once the force of the pool cue is applied, the system calculates the path for the ball. The computer takes into account the simulation update rate (Δt), and at every clock tick of that rate the entity sends out a packet giving its coordinates.

But when the trajectories of an entity are fairly predictable, as these are, a far more efficient method is to use "dead reckoning" [right]. Here the initial coordinates are given at each impetus, as well as the velocity (V) in each spatial dimension. Fewer packets need be sent because when they need to know the location and heading of the ball, the other entities can infer the information by examining the original packet and performing the calculations.

moment. The entity is sent on its way, and at each clock tick of the simulation it sends out a packet giving its location.

But in many cases the behavior of an entity is predictable. Few entities change state continuously, and even then they tend to behave fairly consistently. In the pool-ball example, the path is easily extrapolated and, unless some other event affects it, will not change. A question then arises: the equations of motion are simple, so surely couldn't any other entity, if it wanted to, figure out where the pool-ball entity is and what it's doing? Why does the pool-ball entity have to keep sending updates about its location?

The answer is that it doesn't have to. Rather, some DVEs make use of a technique known as "dead reckoning," which was pioneered in the 1980s in the U.S. Army's Simulation Networking (Simnet) protocol. Simnet has been extended into the Distributed Interactive Simulation (DIS) protocol and, more recently, the High-Level Architecture (HLA) protocol [see "Synthetic soldiers," pp. 39–45].

When DIS transmits entity state information, it includes not only position, but velocity and rotational velocity as well. Thus a roll of a pool ball would require sending only one packet, from which all the hosts could determine the ball's location and orientation at any point in the future (until it reached the edge of the table or hit another ball) [Fig. 2, right].

Usually one specific host somewhere in the network is responsible for the decision-making for a particular entity. In the case of the toy airplane, one computer controls the plane, and that computer is operated by one person. The relationship of the human controller and the thing being controlled can be thought of as a relationship between "pilot" (human) and "drone" (entity). Most of the machines in the DVE network will have only a drone version of the airplane, guided entirely by dead reckoning based on updates received from the pilot.

The pilot host for the plane runs the algorithm for simulating every aspect of the aircraft's flight. It does so in response to its (human) controller's wishes, whether the input is by joystick, mouse, or whatever. The same host also handles the simpler dead-reckoning algorithm.

Only when the results of the two algorithms differ by more than a certain threshold value is a new update sent out to all the drones on other people's machines. in fact, even if there is no divergence, the entity periodically sends out updates on its own status, to let the system know that it is still around. These updates are engagingly known as "heartbeats" or "keep-alives" (in crowded environments, even the constant throb of heartbeats can seriously clog the network).

Dead reckoning not only saves bandwidth, but also makes the system less vulnerable to latency. For instance, since every update message is marked with the time it was sent, that timestamp can be used to compute the current position of the ball on the table if the pool ball's packet arrives late.

From projectiles to dancers

Dead-reckoning is in essence a special case of a more general technique: sending higher- (rather than lower-) level behavior information. In the case of dead reckoning, the higher-level information is the entity's velocity vector; the lower-level information is the result of the equations of motion employing that vector. Recall that the technique was originally used in combat simulation. Velocity vectors are fine for simple trajectories, but for more subtle forms of interaction, particularly with jointed entities (like people), it is possible to define other high-level behaviors, such as walking, jumping, or dancing.

For example, suppose an avatar walks across a room. Initially the avatar might send a packet of information—a packet marked "walk"—giving its starting location, start time, and velocity vector. Here the host is equipped with knowledge of how joints, bones, and muscles move in relation to one another (an active area of research, by the way).

Rather than have the host send a list of joint angles (to describe the motion of the avatar's arms and legs), that information is inferred by, and simulated on, all the other machines in the DVE that base their calculations on the "walk" message and the location and orientation of the avatar.

This method is a kind of data compression. It exploits prior knowledge of the entity's hierarchical structure (feet attached to calves, calves to thighs, thighs to pelvis, and so on) in order to transmit less data. Sending the *x*, *y*, or *z* location of each foot is not necessary because it can be computed from the joint angles and the known lengths of the body segments.

Also note that this DVE software design is strongly object-oriented, an approach that tends to be cleaner and easier to maintain. In programming terms, each entity is an object to which messages are sent. Each class of entity has a repertoire of behaviors that enable it to respond to the messages.

The entity's repertoire is generally created at the same time that the entity itself is designed. In principle, however, there should be nothing preventing other behaviors from being added, from hopping and waving the arms to tap dancing.

A précis of Internet protocols

The Internet is based on a family of roughly hierarchical protocols. The lowest level, called simply Internet protocol (IP), handles addressing and routing. IP packets can be sent over any medium that can move bits: Ethernet, optical fiber, telephone lines, radio links, or tin cans and string.

At a higher level are the transmission control protocol (TCP) and the user datagram protocol (UDP). TCP provides a reliable, connection-oriented byte-stream form of communication. Applications can use TCP to open a logical connection to another host on the Internet, send data down that connection, and then receive data back.

TCP handles the complex chores of breaking the stream of data into smaller pieces, doing checksums on the data to see that it arrives intact, sorting the packets as they arrive to make sure they are in the correct order, requesting retransmission when needed, and handling flow control. On the Internet, the protocols work together: TCP uses IP to actually move the packets to other hosts.

TCP is the protocol upon which most Internet applications run. The World Wide Web uses hypertext transport protocol (HTTP), which is sent over TCP. When a user clicks on a Web link, a connection is opened to another host using TCP; the address the link points to (a universal resource locator [URL]) is sent down the wire, the data is received back, and the connection is closed.

Similar techniques are used for Unix programs like Finger (which finds information about the users of machines on the Internet), simple mail transport protocol (SMTP—for handling all electronic mail over the Internet), and network news transport protocol (NNTP—for Usenet). In all cases, the distribution of data is handled transparently by TCP.

The other higher-level protocol, UDP, differs from TCP because it provides a connectionless and inherently unreliable way of sending data from one host to another. "Unreliable" in this context does not mean that UDP is somehow error-prone; it just means that the arrival of a specific packet is not guaranteed. If it does arrive, though, it will arrive intact.

UDP has some advantages over TCP. The problem with TCP lies in its strengths. Because it does all kinds of flow control, error-checking, and sequencing, it has a fair amount of overhead, which translates into undesirable latency.

In contrast, UDP packets are relatively "lightweight." Although they get lost on occasion, they will not clog things up. Also, unlike TCP, UDP packets can be broadcast on subnets (LANs), because no logical connection is involved (recall that UDP operates in a fire-and-forget mode).

All the same, because UDP is unreliable, in the strict technical sense, it requires what are known as stateless protocols: each message must be complete and self-contained, and must make no assumptions about previous messages having been received.

UDP is used by several common Internet programs, including network time protocol (NTP), which handles the synchronizing of clock time across the Net.

Also using UDP is the military's Distributed Interactive Simulation (DIS) protocol, which is stateless for the most part. That is, UDP broadcasts packet updates to hosts, and each of the updates contains the complete state of the entity. Each is re-broadcast every few seconds, or more often if the entity determines that more frequent updates are needed. —*B.R.*

A consistent view of the world

One of the trickiest tasks in designing a DVE system is maintaining a consistent representation of the world on all the different hosts. If multiple users try to interact with the same entity at the same time, who "wins"?

A number of approaches can be used to solve this problem. The simplest way is to use the real-life social protocols everyone is familiar with. Users of a DVE stay aware of what other users are doing and politely allow them to go ahead. If someone indicates that he is going to open a door, no one else interferes. This arrangement works fine in most situations, but there are some application areas in which users cannot be relied upon to follow this sort of protocol—combat simulation or on-line gaming, for instance—nor would anyone want them to.

Other approaches have the software negotiate conflicts. The entity itself may arbitrate access. For instance, a user wanting to pick up a suitcase sends it a message and the suitcase agrees to respond. If multiple users grab for the suitcase simultaneously, the suitcase itself decides who got there first.

This approach works well, but can slow down the system. If a user wants to carry the suitcase across the room, she needs to constantly get its permission. If it takes 250 milliseconds for a message to get from the user to the suitcase entity, and another 250 ms for a response to come back, there will be an annoying half-second of delay whenever the entity is manipulated.

Another approach is locking. Upon being picked up by a user, an object announces to anyone else attempting to manipulate it that it is "locked" by that user. For the object to change state in any way (be moved or recolored, say), that change must come from its owner/user.

Difficulties with locking can arise if a user abruptly goes off-line without releasing the lock, since the entity would in effect be "orphaned." One solution to this problem is to have the entity's user/owner periodically send packets verifying to the locked entity that its owner is still there and in control. If these packets are not received, the entity unlocks itself.

Still another technique is attachment. Here, if a user picks up an entity, the entity announces that it is "attached" to the user. The entity retains control over itself, and may detach itself from the user at any time. The entity's position and orientation, however, are all relative to the user, so carrying the object across the room does not add to network traffic.

Filtering updates

Although the technique of sending just higher-level behaviors does use less bandwidth, it is still not enough to allow for the data that must be passed around. A large-scale DVE might have hundreds or thousands of active entities, so even if each entity sends out a packet only every couple of seconds, the available bandwidth will be quickly exhausted.

For example, even if each update message contains the *x*, *y*, and *z* coordinates to which an object is moving, as well as the rotation around *x*, *y*, and *z* axes, a timestamp giving the desired arrival time, and a 4-byte identifier, the packets still ends up 32 bytes long. Even if a packet is sent only every other second, the bandwidth averages out to 16 bytes per second per entity. If there are a couple of hundred entities, the available 28.8-kilobaud data rate is again exceeded.

For further reductions of data, useless information can be filtered out of the update stream. One filtering technique is to calculate the distance that a user could "see." An entity beyond that (simulated) distance need not send its status to that user's avatar.

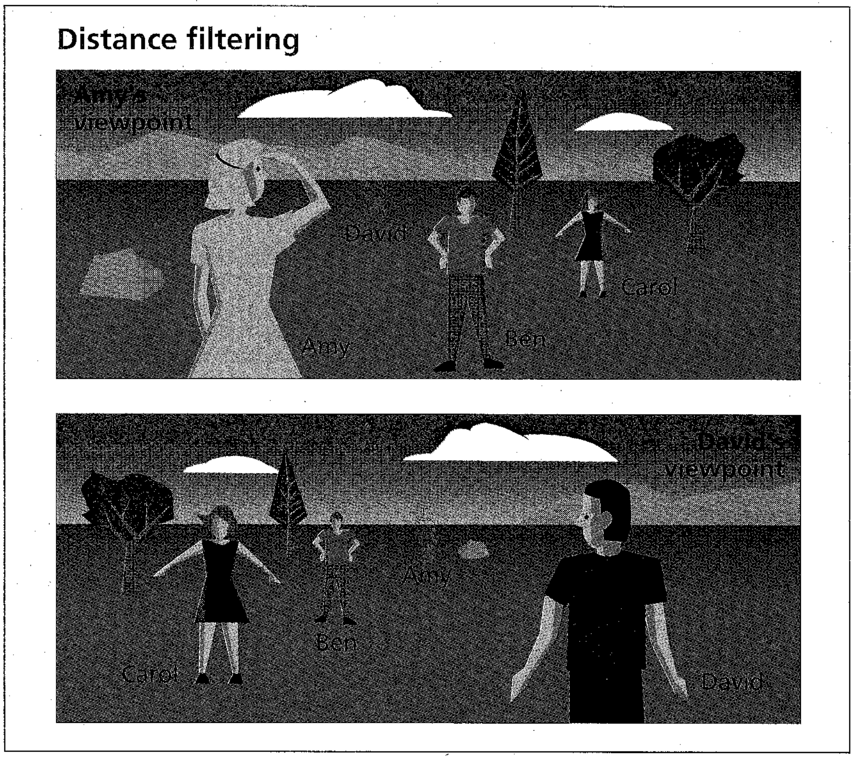
Note, however, that the "far-away" entity cannot stop sending packets—it may still be perfectly visible to other entities [Fig. 3]. Calculating whether entities are in range of each other, and if so, whether they are visible, must be done between each pair of entities. This example uses visible light and models of human vision, but the same principle would equally apply with different parameters—in a gaming DVE to a Superman that sees through walls or in a military DVE to an infrared seeker. The technique bears some relation to what in computer graphics research is known as level-of-detail, in which the number of polygons representing an object (the amount of detail) varies with the viewer's position.

The simplest filtering technique of this kind is based on distance; if the ratio of an entity's size to its distance from the user is less than some predetermined threshold value, then its actions are not currently relevant to the user. A bird in flight 20 kilometers away cannot be seen, so sending its update packets down the wire is unnecessary.

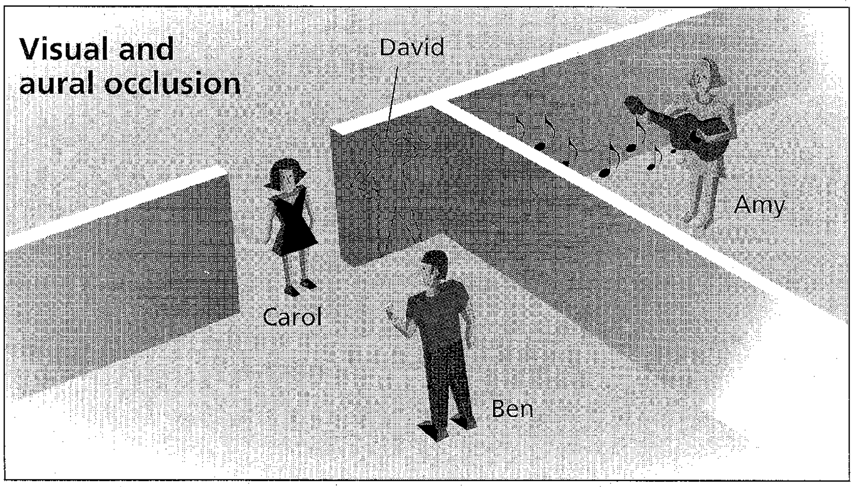
Some other entities may be obscured by fixed features in the environment, or by the presence of certain environmental effects (fog, clouds, and the like) in the modeling software. With a technique known as occlusion filtering, updates for such entities would not be sent to the user's machine, since doing so would use up precious bandwidth and processing cycles.

Another form of relevance filtering uses spatial subdivision. The virtual environment can be divided into zones and zone-to-zone visibility can be precomputed. Suppose an avatar is in the bedroom of its virtual home; it will be unable to see anything in the kitchen, garage, or den. If the DVE system knows this, then it will not send it any updates for entities contained entirely in those regions. Sound occlusion is not so clearcut a process, however, because the user may be able to hear events in the kitchen [Fig.4].

For the most part, distance-based filtering is useful for outdoor environments while occlusion-based filtering is useful for indoor ones.



[3] To contain the flood of data packets, many DVEs use filtering techniques to analyze the relevance of data before passing it on. In one type, distance-based filtering, the system calculates on the basis of stored information whether a particular entity will be in view. From Amy's point of view [top], data packets are exchanged between her and everyone but David, who is too far away for her to see (deduced from real-world experience). But the calculations must be performed for each pair of entities: David is still very much an active entity, sending packets to the people he can "see."



[4] Occlusion filtering is another way to lessen the amount of data flowing through a distributed virtual environment. It builds on the fact that people ordinarily cannot see or clearly hear through solid objects. When a host computer analyzes the location of the walls and the position of the figures, it notes that David, like Amy, is visually cut off from everyone else, so the two do not need to send visual data at all, and that Ben and Carol need send visual data only between themselves (occasionally David does send "keep-alives," data sent to the system signaling that he is still in the scene). Amy sends sonic data packets to all.

Decentralizing the system

Choosing the high-level network protocol is important in DVE design—on any kind of network, and not just the Internet. The choice is between a connection-oriented approach, such as transmission control protocol (TCP), or a connectionless approach, such as user datagram protocol (UDP) [see "A précis of Internet protocols," p. 35]. TCP provides guaranteed packet transmission, but at the cost of high latency, while with UDP state information must be re-sent periodically to ensure that none of the hosts lacks crucial information. Each has its pros and cons, and most DVEs use some combination of connectionless (UDP-like) and connection-oriented (TCP-like) protocols.

Another cardinal aspect of DVE design is the topology the network uses. A naive network might have a single central server to which all the client hosts connect; the server at the hub would relay packets between hosts, doing relevance filtering as needed [Fig. 5, left].

Such a system works fine for toy worlds with a couple of dozen participants, but clearly that world will not scale up. As the number of simultaneous users rises, the central host will become overburdened, because more memory and central processing unit (CPU) cycles would be needed to cope with the additional entities.

Using a more powerful computer for the central host is possible, but the available computing power will never increase as rapidly as the peak number of simultaneous users does. Sooner or later, the central server will no longer be able to handle the load. Even were unlimited quantities of memory and boundless CPU power available, the sheer volume of data flowing into the hub will unfortunately start to saturate the available bandwidth.

An alternative approach establishes a direct TCP connection that links every host to every other host, eliminating the need for a central server [Fig. 5, right]. But a new problem arises: each host must send every update $n-1$ times (once for each of the other n hosts it's connected to in the simulation). This setup also does not scale, and sending the same update message several times is a waste of bandwidth.

The direction this technology takes will not be determined solely by technical criteria, but by market considerations. In spite of the technical advantages of decentralized, highly distributed virtual environments, many commercial DVE firms are creating server-oriented systems. The reason is simple: their business model is based on giving away the client (host) software to users, but selling the server software to those who want to create their own DVEs. Without large central servers for their clients to connect to, DVE firms have no revenue-producing product.

Still, there may well be ways of generating revenue from a fully distributed virtual environment. After all, the World Wide Web is entirely decentralized, yet Internet service providers such as America OnLine are determined to turn a profit. It remains to be seen whether a centralized or decentralized approach becomes prevalent.

A multicasting alternative

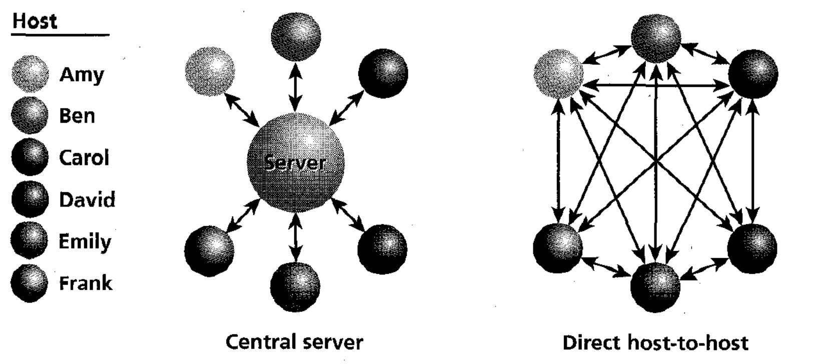
Recently, several groups have had good success using a communication scheme known as multicasting. In multicasting, a packet is addressed not to an individual host, but to a group address. The group address does not point to (reside at) any particular host, any more than a conference telephone call is taking place at one phone. Bottlenecks are lessened because each host can choose whether to listen to what is coursing through a particular multicast group address at any one time.

This approach should not be confused with broadcasting, in which packets are sent to every single machine on the relevant network. While similar to broadcasting, a multicasting system makes it much easier to reject packets so that far less of a burden is imposed on every host in the network.

Multicasting has a number of pluses. It requires no central host, has no need for a host to send an update packet more than once, and it scales well.

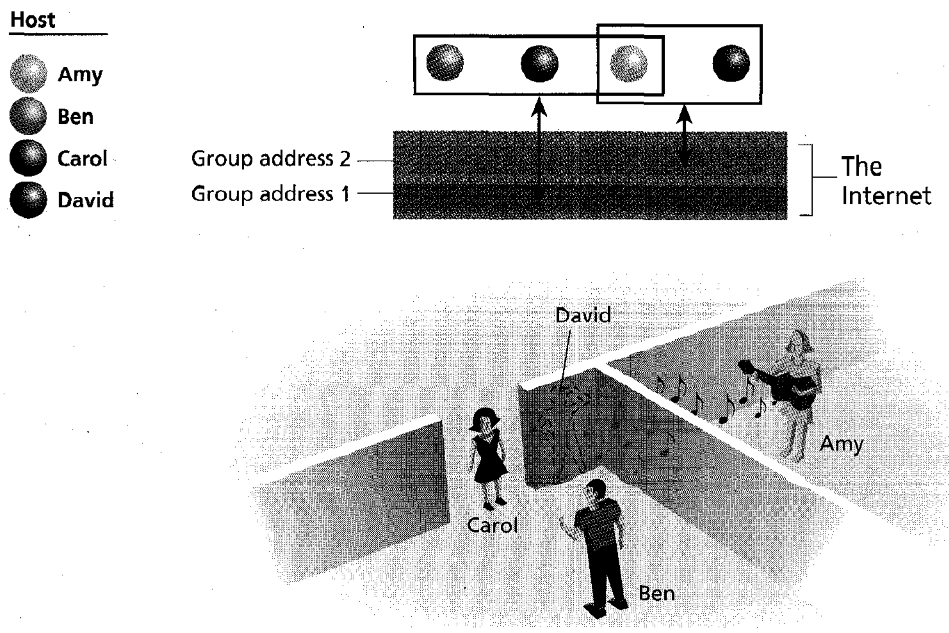
Perhaps most importantly, it also maps well to occlusion-based filtering, as described above. Here, each occlusion zone (comprised of a set of entities in virtual contact with each other), whenever it changes, corresponds to a multicast group address created on the fly. A host joins or leaves only those multicast groups that contain data based on which subsets of the overall environment it can see [Fig. 6].

Unfortunately, multicasting protocols are not yet widely supported, because few network routers can handle multicast packets properly. There are limits, too, on the number of multicast addresses available in the current Internet protocol (IP). Multicasting is also suitable only for connectionless protocols (such as UDP), since no "connections" as such are actually maintained (a multicast group address can be thought of as a self-contained data high-



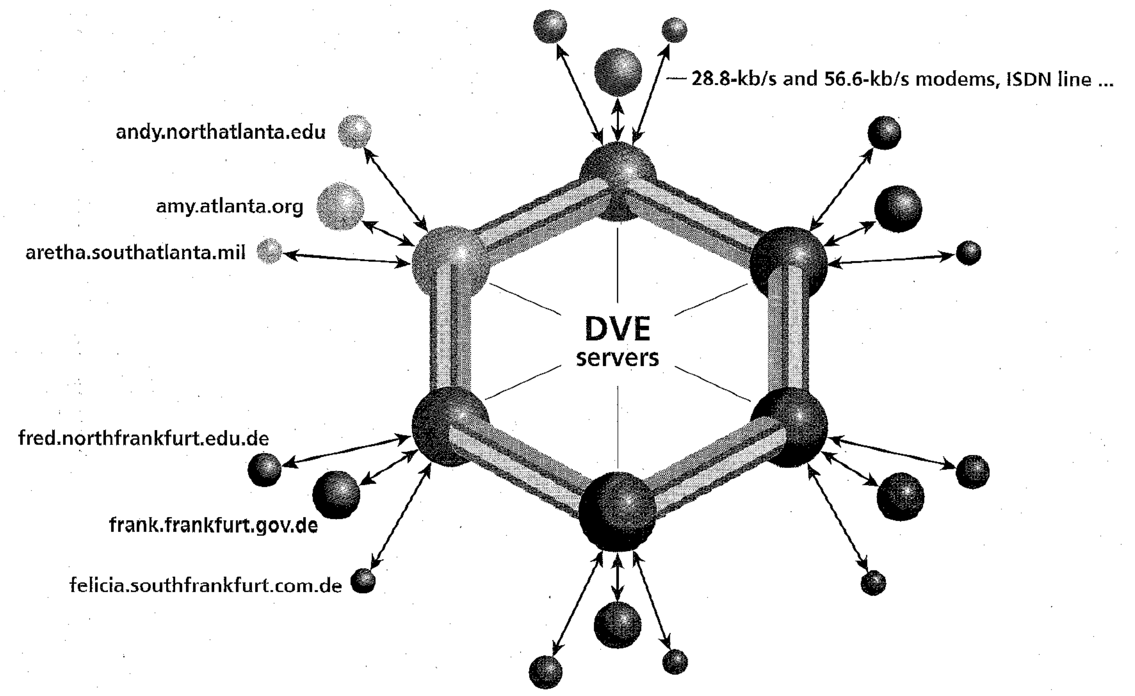
[5] A simple network with a central server fails as the number of users (hosts) steadily increases [left]. The processing power of the server would be unable to handle the load—and even were unlimited power available, the bandwidth would be saturated.

An alternative configuration that eliminates the central server leads to another problem [right]. A direct connection exists between each of the hosts. But since each update must be sent $n-1$ times (where n is the number of hosts), the connections are clogged with duplicates, and much of the bandwidth is wasted.



[6] Multicasting is an efficient protocol for DVEs because only the data needed for particular hosts at any one time are made commonly available to those hosts. The data are sent by way of an Internet group address, which is created on the fly by the system software as the data needs of the users change. At this moment[[also depicted in Fig. 4], Ben and Carol are in sight of each other but no one else, and both can hear Amy's music; the three hosts need to send visual and sonic data packets to each other only, doing so by way of an Internet group address 1.

Concurrently, although he is visually occluded from everyone, David can hear Amy's guitar. The specific data necessary for just these two users are made mutually available by way of Internet group address 2.


28.8-kb/s and 56.6-kb/s modems, ISDN line ...
andy.northatlanta.edu
amy.atlanta.org
aretha.southatlanta.mil
DVE servers
fred.northfrankfurt.edu.de
frank.frankfurt.gov.de
felicia.southfrankfurt.com.de

[7] In the author's proposal, central-server and direct host-to-host network topologies are combined (in this case, DVE servers themselves are considered hosts). In the example shown here, the DVE servers, scattered around the world, are directly connected by multicasting [only two multicast data highways are shown for simplicity]. Each user (such as Amy and her physical neighbors) would normally connect to the central server that is geographically closest, to lessen the number of hops the connections must make. Users could connect to the server with dial-up modems, integrated-services digital network (ISDN) lines, or whichever technology they choose. As an added benefit, the DVE server to which the user connects might well be co-located with her Internet service provider, which need send only relevant updates down the relatively low-bandwidth link to her machine.

way). The next major revision of the IP protocol, called IPNG (IP–Next Generation, or IP6), will have support for much larger address spaces and improved support for multicasting.

The hybrid way

A network approach that looks promising and has undergone a good deal of research is a hybrid system. Users would connect to a DVE server, which would handle relevance filtering and similar chores [Fig. 7]. In this setup, each DVE server would be connected to other DVE servers using multicasting. These servers would guarantee that the software running in the client machine neither knows nor cares what technology is being used between servers, in this way ensuring compatibility among various systems. The local networking technology need not be the same for all of the clients—some users could connect to the server through direct integrated-services digital network (ISDN) connections; others might use dial-up modems.

The hybrid approach is a good match for the modems and ISDN lines which people will most likely have for the near future. Furthermore, the DVE server to which the user connects might well be co-located with her Internet service provider (this distinction is not illustrated in the figure). With this arrangement, only relevant updates need be sent down the low-bandwidth link to her machine.

Note that none of the topologies described above necessarily has any relationship to the low-level network topology. Regardless of whether the underlying network is Ethernet-based, token ring, or some other approach altogether, each of the above models applies at a higher level.

Putting the standards in place

Clearly challenges still lie ahead for DVE systems, but in most cases solutions seem to be close. By sending higher-level behaviors, doing relevance filtering, using connectionless protocols, and working with a decentralized architecture, bandwidth and latency problems can be resolved. A variety of approaches are also available to maintain consistency across hosts.

But what is clearly needed, and needed soon, is a set of standards. The Virtual Reality Modeling Language (VRML), deals with how three-dimensional virtual objects can be designed for transmission on the Internet. Now at version 2.0, it is soon to be an ISO standard. It then will be extended so that objects in those environments, and the environments themselves, can be shared in real time.

A working group called Living Worlds is putting together a proposal for multi-user standards for VRML. A related proposal called Open Community, for an application programmer's interface for multiuser technology, has been announced by Mitsubishi Electric Research Laboratory and a number of other companies [the complex issues in providing support for DVE developers are detailed in "An infrastructure for social software," pp. 26–31].

It is not overstating the case to say that once these standards are adopted, this new version of VRML will have the same impact that the Hypertext Markup Language (HTML)—the World Wide Web's protocol—had on the Internet. ◆

About the author

Bernie Roehle is a software developer based in the electrical and computer engineering department at the University of Waterloo, Ont., Canada. He is the author of *Virtual Reality Creations* and *Playing God: Creating Virtual Worlds*, published in 1994 and 1996, respectively, by Waite Group Press, Corte Madera, Calif., and, with co-author Stephen Matsuba, *Special Edition: Using VRML* (Que Publishing, Indianapolis, Ind., 1996). Next month his book *Late Night VRML and Java* will be published by Ziff-Davis, New York. He chairs the Virtual Humans Architecture Group.

Spectrum editor: Robert Braham