An overview of multiplayer gaming

by Nicolas Brodu
Concordia University PhD Student
07 October 2004

Introduction

We've been programming them for years, why are multiplayer games still a problem?



The first multiplayer game can be retraced back [1] to SpaceWars, 1962. It consisted in 2 players on the same machine, controlling a rudimentary 2D output. Three decades later, with the release of Doom in 1993, 4 players could challenge each other in a 3D environment, on a local network. Ten years later, massive multiplayer on-line games gather more than 100 000 simultaneous players in virtual worlds, a new feat possible thanks to the Internet.

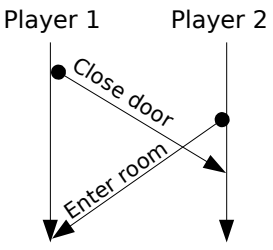
No need to say that as time passes and game development becomes more and more complicated, needs change and new problems appear. This overview presents the main challenges as multiplayer gaming complexity continues to increase, and some possible solutions. It is by no means a deep analysis of each of the issues, some of them would need whole books to make them justice. The goal is to present the main issues, sketch possible solutions, and provide references pointing to deeper analysis.

Global time does not exist

This is a well known problem in multi-thread and distributed system programming, and it is summarized briefly by schema 1.

This fundamental issue has deep consequences, which can be classified in three categories: Synchronization, Scalability, and Security. All three aspects are interrelated, and as we'll see later on, addressing one particular issue impacts the other two.

This diagram shows the main problem appearing when two contradicting events are emitted at the same time: The local order of events is reversed, leading to a different game state. A variant of this problem appears even in the case of a single machine with concurrent threads, and something needs to be done to maintain the system consistency.



Schema 1: Global time does not exist

Synchronization is the direct consequence of having no global time, and is the difficulty of sharing a game state consistently across participants.
Scalability is how well a solution to address this problem adapts to the number of players. An algorithm may work well for a few players, but not for a few hundreds, or thousands.
Security is about the prevention and detection of cheats. Flaws in the solutions to ensure synchronization are common source of security breach, but not only.

Solving the global time problem is harder than it seems, especially for its impacts on security. Even the TCP protocol, described first [2] in Sept. 1981, was found [3] to have a little flaw in April 2004. It is related to packet sequence numbers, which were introduced precisely as a mean to synchronize received packets. This flaw could allow a denial of service attack, which means it could block a server from providing its services to legitimate clients. Fortunately, taking advantage of this is tough. Unfortunately, previously discovered flaws [4] allow easier attacks...

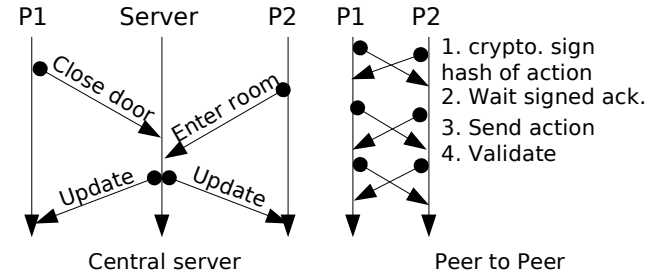
As a conclusion, no protocol or application in an open environment should assume things like constant response time, correctly formatted packets, etc... In practice, no perfect solution was found. Thus most applications actually accept the risk of an attack on the underlying transport mechanism, including games, as does the other parts of this overview.

Scalability

Now that we've in mind no solution is perfect, let's go back to scalability and possible ways to overcome the fundamental problem of synchronization. Scalability is related to maintaining a consistent game state, despite all the problems that can happen [5].

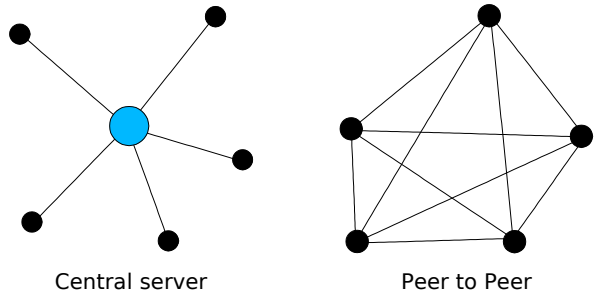
When dealing with a single machine configuration, techniques like mutex and semaphores can be used for local synchronization. The problem really manifests itself on a networked environment. In this case, event synchronization is still important, but distributed data across the network should also be taken into account. And to do so requires addressing bandwidth limitations, possible network failures and data duplication during reconciliation, etc. Of course, techniques like data compression and packet aggregation

[6] may help for data transfers, but are not enough. What needs to be done is a global treatment of scalability, by designing the system with efficient algorithms.



Schema 2: Possible synchronization solutions

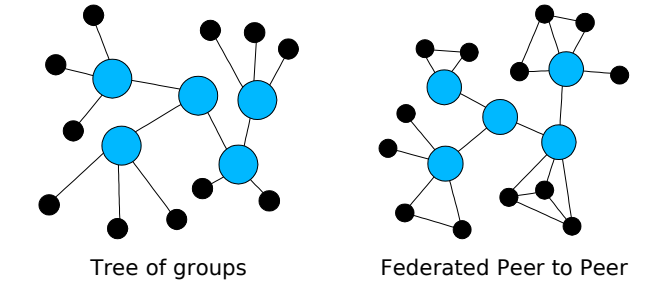
Schema 2 presents two common configurations. The simplest solution is probably to gather all events on a central server, and use its unique time reference as the “true” one. The server can then solve conflicts, and send updates to all participants. Unfortunately, this is unfair in the case of unsymmetrical connections to the server. In this case, the machine with the smaller network lag will always get priority on the others. Another problem is the network latency: a full round-trip to the server is necessary to get the update for an event, which greatly reduces interactivity. Finally, algorithms using a central server are by essence of O(n) complexity, which only scales up to a limited number of players. This architecture is thus a favorite choice for games with the order of 10 or 20 participants in a LAN, given its simplicity. See also the appendix for a possible solution to global synchronization using a unique time manager, while maintaining some level of performances thanks to parallel processing.



Schema 3: Network architectures for schema 2

The most straightforward way to suppress the bottleneck at the server is not to use a server! The leads to the peer to peer architecture, as shown in schema 3. Since there is no central server to sort the events out, the peers need a trustful mechanism that prevents repudiation and other forms of cheating. The following 2-pass algorithm, sketched in schema 2, is a variation of the lockstep algorithm presented in great details in [7]. In a first pass, all participants sign a hash code of their intended action, and wait for other participants to acknowledge the time and hash code. Then, in a second pass, the action is sent. Since all participants can now verify the validity of the hash code, and they had a chance to synchronize before acknowledging, it is possible to maintain a consistent game state. This solution has the great advantage of improving security, since no-one can wait for another player action and pretend it has already engaged in a counter-measure by forging a packet with a previous date. Unfortunately, this algorithm complexity is $O(n^2)$, and it does not scale at all.

The only really tractable complexity a game can afford is O(log n) [8]. A natural way to achieve this is to use a tree-like structure, where nodes for each player are grouped according to some internal game parameter. Schema 4 presents such an architecture. This solution may also be used for global synchronization, with increased latency for leaf nodes proportional to the tree level, compared to the central server solution. Hopefully, it is usually possible to assign nodes to logical groups in the game. For example, if players are gathered by geographic area, there is no need to synchronize the different groups together, because players in each group don't meet in the game at this particular time (groups dynamically change when players move). Thus, in this case, global synchronization is not necessary anymore.



Schema 4: Advanced network architectures

Finally, within a single group, it is possible to reduce the central server bottleneck. Take for example the very common case of an on-line game in which players can customize their character appearance. If each player node had to communicate through the server to get the other players appearance, the server would pipe back all the graphical data, and would end up doing too much of this kind of communication. Instead, it is much better to let the players exchange their graphical data directly, and keep the server resources for more important data and events: the server is used only when a trust relationship is needed (ex: damage, player life, contradictory events...), and player nodes can exchange less critical data directly (graphical appearance, in-game player conversations, etc.). This architec-

ture is called federated Peer to Peer, and shown in the schema 4. It was studied in details in [9].

This last architecture has the nice advantage of keeping the global algorithms complexity order close to log n, provided the local peer to peer groups aren't too large. The following section deals with this issue of how small a local group can be while still preserving consistency.

All in all, the conclusion on scalability is that design and network architecture are the most important factors. Technical choices like reliable vs unreliable protocols may help, too, but on a large scale the network architecture is the main point to consider.

Interest Management

This is the art of defining local areas of interest for each entity, and achieving synchronization only in intersecting regions of interest. How the synchronization is achieved and how each local region is defined are still research topics (see [8], [10], [6] and [11] for example). This discussion is by no means a thorough covering of the subject, but a rather simple personal version presenting the main idea.

As pointed out in [8], so long as players at the same location share enough information about their common environment, they have the illusion of being in the same virtual world. It doesn't matter if a few details differ: players won't ever notice that other players have a slightly different view.

With this idea in mind, let's note that:

- An action at one place affects only its neighborhood, even the same geographical group, and does not need to be propagated to all players of this group: this is a really local synchronization.
- The locality is defined in the game context. Physically the nodes may be in very different places, and the best network architecture for game locality may have to be adapted to cope with physical limitations (by setting a game server on a backbone close to the client for example [8]).
- Locality has a different definition for each object. In a real-time strategy game, or in a military simulation, a radar object has a much wider domain of interest than an infantry unit. Hopefully, a wider interest generally means a lesser level of details: in the previous example the radar may only care for the position of objects in its range, and no other parameter. Thus, the overall data volume may still be tractable.
- Corollary of the previous point: regions of interest are not symmetric. A player hidden behind a tree or a stealth unit may act on another player, whereas the other player shall not even be given the chance to get the information that something is hidden for security reasons.

Description

Only one instance of each item exists in the game world. There may be clones sharing the same description, but there will be a unique ID for each clone. Specific methods for attributing and maintaining the ID to object mapping are out of scope of this discussion. An implementation using a distributed hash table is described in [11].

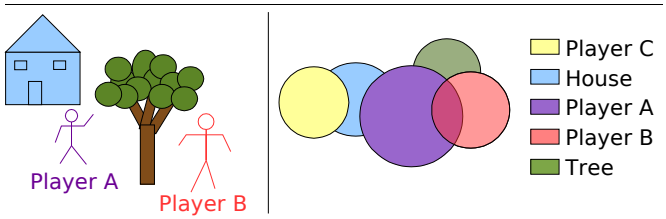
Each node is then responsible for a few items, and serves as the reference for other nodes updates. Techniques exist to improve performances, like a local caching mechanism to query details only when an ID changed. Again, the reader is invited to check [8], [10], and [11] as starting points for further investigations on the subject, as we'll focus here on the main idea.

In a federated peer-to-peer environment, the servers maintain internal game information (items, player characteristics...) while the unsafe user nodes may maintain player-related unauthenticated information (visual appearance of the avatar, message log, etc.)

When a player moves in world, the player node asks neighbor nodes for more information concerning new elements. The relevant parameters are compared to their cached values, and if necessary the decision is taken to incrementally ask for more details as needed.

An example

An example of regions of interests is presented in schema 5, based on an hypothetical game scene. Player A enters a city. The node asks for the game server neighbor elements (in the field of view with a max distance): it gets back object IDs for houses, trees, other players, and their respective positions in 3D. Suppose only player B is visible, below a tree. Node A then asks node B directly for the player avatar models & textures, and the server node for the model and textures of the decor elements (house and tree). Node A does only ask details at the precision required by the object distances. Advanced continuous level of detail techniques may be used at this point to further minimize the bandwidth usage, see [12] for example.
As player A moves to see player B under the tree, a more precise version of player B character model and textures is requested, together with tree details from the server. Depending on the importance of this particular tree and the



Schema 5: Regions of Interest, in-game example

customization made by player B, their nodes may very well return nothing. In this case, the player A node can compare the item IDs it wants with the ones in the game local catalog. The tree will probably be generic, and the high-resolution texture available from the local cache. But it was worth asking in any case, since the tree may not be a generic one. The same reasoning holds for the other player avatar: without customization the model and textures are drawn locally from the standard game catalog, otherwise only the differences are retrieved from the other player node.

In schema 5, the ellipses represent regions of interest. In this example, only player A and B have a symmetrical relationship. The tree may not act on player A, but player A needs to get the tree model and texture. Similarly, the internal state of objects needs not be transfered until a relevant action is engaged by one player. [11] gives an example: "A chest in a dungeon must communicate its location and appearance to players, but not its status as locked or unlocked, or its content". Together with benefits on network load, this significantly improves security.

Coming back to the player illusion of being in a common world and local synchronization, it is worth noting that continuity between regions of interest is ensured by the environment. In the schema 5 example, a player C may be in the house. In this case, players A and C do not initially have intersecting regions of interest. If player C comes out of the house, both players may somehow become synchronized thanks to the common house node. Thus, passive elements serve as links to ensure a connected topology. Even better: player C has discovered player A directly, without any kind of broadcasting, and without having to maintain all player positions on a centralized server.

Interest management offers serious advantages:

- No single player node needs to contain the whole world: The game can be shipped with only the bare necessary minimum, together with a local catalog of commonly used big media files to make an initial cache for the world items. As the player discovers more of the game objects, their description is downloaded directly from the official server and other player nodes. Thus, the game can evolve even after it is released.
- Parts of the world where there are no player fall back to the official servers permanent storage, and will be delivered on request. No run-time resources are wasted so as to handle unused parts of the world. This also works for introducing new items, objects, monsters, etc. without causing inconsistencies.
- Synchronization is achieved at local level: players at the same location share information about their common environment and have the illusion of being in the same virtual world. A few details may not match, like for example the texture of that house in the background, which is closer from player 1 than from player 2. It doesn't matter, since in this case the full-version will be downloaded by node 2 when player 2 comes closer (and it may even be downloaded from node 1 to ease the server load, possibly by checking on the server digital signature to certify player 1 did not tamper with the data).
- Minimum traffic thanks to caching and incremental transfer.

But it suffers from some drawbacks

- It's a complete overkill for simple LAN games, in which case a simple central server architecture should be much faster.
- When delegating part of the game processing to player controlled nodes, the game makers have to be extremely cautious about malicious nodes, and security issues in general.

Graceful Degradation

To paraphrase [8]: perfection is not possible, avoiding failures is already good, but graceful degradation is even better!

The idea is to have successive levels of failure, so as to avoid a complete crash when an unexpected error occurs.

For example, we mentioned earlier that network latency causes players' view of the world to be imperfectly synchronized, but for short periods of time this may be acceptable. It may even not be necessary to reconcile all the differences: A fast-moving player node won't download all the details as the player moves away before having a chance to get them, but when she stops then the details for the final location are flowing in. In this case, it is better to accept the small degradation of imperfect synchronization for details, rather than use a reliable protocol for all the data. This last case would effectively work better on an ideal network, but wouldn't work as well on a real network where less important data would be given equal treatment to critical one.

Each object thus has its parameters classified by consequences in game of not being exact. The

most important parameters are provided first, and the others are downloaded incrementally if they don't become irrelevant by that time. In the case of missing or late data, default parameters are used to provide an acceptable degradation: If a specific wood texture can't be downloaded for the newly discovered object, falling back to the generic default wood texture in the local player database may be enough. In the case when even this is too slow, then rendering a flat-shaded brown may work equally well for temporary scenes.

What's true for data is also true for events. The idea is to separate 2 kinds of events based on their synchronization needs:

- Soft synchronization: The event may be reconciled later without immediate consequence for game play (ex: player 1 overtakes player 2 in a race game).
- Hard synchronization: The event should be synchronized whatever the cost (ex: crossing the finish line in the same race game).

As for data, in some cases a soft event may not need to be reconciled, if another event makes it obsolete later on.

Finally, the idea presented in appendix for civil and military applications, posting events in the future so a short period of time is predictable, can be re-used to some extent. For example, being able to estimate in advance the bandwidth needs, even at a the local level of a few objects, allows for prioritizing requests so the most important data gets through first. The prediction may not be exact if another object interferes with the local group, but the assumption is that most of the time it will lead to an optimization.

The conclusion for this section is that designing the system to include failure management in its core is much better than having to deal with errors on a case by case basis.

Degradation Recovery

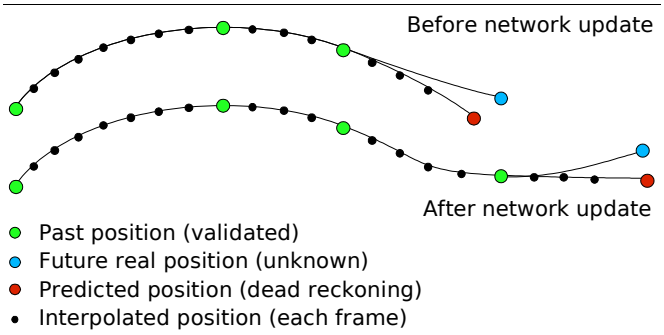
There's no replacement for missing data. Unfortunately, knowing this is for the sake of global performance (like when using an unreliable protocol) or the prioritizing of other more important data (as seen in the previous section) offers little compensation. One possible way of handling this situation is by using default parameters, as mentioned above, but there are cases where this technique can not be applied.

The simplest degradation coming from network lag is one of them. What to do when a frame needs to be rendered each 40ms, but data comes from the server each 200ms? If some data is missing or arrives too late, is it still possible to provide the player with consistent values, and do the reconciliation later?

Dead reckoning

Dead reckoning is a technique to achieve this. It consists in a prediction based on current values. Many articles were written on the subject. As before, we'll stick to the main idea in the presentation, and the reader is invited to check [5], [6], [13] and [14] (for example) as pointers to more information on the subject.

Usually some data can be extrapolated, the most common being position. Using the current values of a player speed and position, it is possible to compute an approximation of the next position. In the case of a player moving in a straight line, the position may even be correct with a first-order approximation. So, if some data is missing, simply using the predicted value may be a good enough guess for a graceful degradation.



Schema 6: Dead reckoning

The problem is at reconciliation, when the real data arrives. In the case of position, the player may have turned around and gone on one side for example. But a good game already interpolates the position between successive dates, if only for creating a smooth movement at each frame instead of an instant jump from one position to another. It is then just a matter of more interpolation to get from the last point predicted using the guessed position, to the next real updated position.

[6] and [13] propose using cubic splines, since it allows for smooth transition even in the case of fast second-order “jerk” changes in acceleration (like when abruptly turning in another direction). Other interpolation proposals and their detailed analysis can be found in [15].

Now that we have a good prediction algorithm, why not take advantage of it? The idea proposed in [16] is to transfer data only if the new values are too far from the prediction. Since the sender knows what value the recipient will compute as a prediction, it becomes useless to send data that will be guessed correctly. This may indeed save

bandwidth in some cases, especially if the prediction is systematically computed, in which case there in no extra CPU overhead.

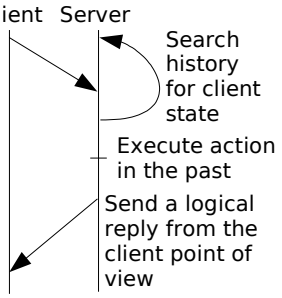
[14] presents valuable practical comments on dead-reckoning from Valve Software, together with information on potential security risks, and the unfairness inherent to this method.

Latency compensation

Another technique presented in the same article [14], in the case of a central server game, is for the server to compensate each player's lag by storing a short history, and compute the point of view of the world for each player particular time reference (see schema 7).

The idea is that players base their decisions on what information they have locally, and may have acted differently if they knew what would happen later on. Executing the player decision when it is received by the server is not right, because of the small delay due to network latency. To be fair, the server should execute the commands it receives at the date they were emitted, to respect the players decisions.

Since players base their decisions on the information they have locally, the server goes back in history to compensate the network latency. It then executes the action in the past, propagating changes if necessary. Players decisions are thus respected, and they have the impression there is no lag at all. The unfairness of network latency for every player is traded for some unfairness in executing actions in the past.



Schema 7: Latency compensation

Combined with dead reckoning, this effectively gives the illusion of immediate response for each player.

Unfortunately, since all the actions are now executed in the past, there may be undesired side effects. For example, suppose player A has a better connection than player B. Since the server respects each player local time reference, it might happen that player A had the time to put herself in a position safe from player B. More specifically, the “magic bullet” example in [14] is as follows: player A turns a corner while player B is shooting from behind in a corridor. In this example, from player B point of view, player A is right in front. The server records a hit. But in player A point of view, player B was left behind the corner, and A has the impression that B can shoot around corners!

Hopefully, this is an extreme example, and most of the time the side effects are unnoticeable. The same example with player A and B crossing each other would be accepted by both players. In this case, the side effect results in a slightly wrong angle for player B shot. Since the human mind can't make the difference between 90° and 85° in a short time, player A accepts the outcome.

Thus most of the time, the unfairness of network latency is compensated without any noticeable adverse effects. All in all, both techniques (dead reckoning and lag compensation) have their use, and should be combined for the better gaming experience.

Security

Security is not a problem for a few persons in a private LAN, trusting each other. It's also not a problem if players in a private LAN decide on a common basis to change the game experience with clever cheats, so long as they enjoy themselves. After all, it's their own copy of the game, so they may have fun with it any way they want. Problems arise in environments mixing honest players and cheaters. The game credibility is at stake in official tournaments. In a massive multiplayer on-line game, a cheater affects the game experience of thousands of other players. Some of which become unhappy, disgusted, and leave the game. And as if bad game reputation from former players is not enough, the game makers will often take the blame instead of the cheaters, for not having been careful enough.

An overview of security issues can be found in [17], and an excellent analysis in [18]. A lesson learned from cryptography is: security comes with a good design, and never from obscurity. This is also true for games, and the following discussion is an overview of some common cheat sources, with possible solutions, if any.

Network latency

Network latency can be exploited in several ways. First is the well-known denial of service attack. In this scenario the cheater knows the network address of the victim, and has a much better connection. The cheater then floods the victim network to the point where its latency becomes unacceptable. Usually, in the case of a central server environment, the victim is then disconnected so as not to slow the game for every other player. This way, a cheater may disconnect a winning victim from the game.
Prevention of this attack is unfortunately hard. The game may adopt heavy degradation techniques and continue providing the victim a lesser level of service, though this probably still gives the cheater an advantage. Another possibility is to monitor the network lag throughout the game,

and to make statistics on the number of wins by disconnection. Players may refuse to engage in a game with a potential cheater by checking on these statistics. Of course, this is of little use for single-event games, or when creating a new user account allows the cheater to change identity.

Other attacks related to the network latency are based on the techniques used precisely to overcome it. [7] shows that dead reckoning is inherently unsafe: if a cheater knows what the prediction algorithm is, it is easy to compute another player's current view. This knowledge could give the cheater an advantage, especially combined with a scripted action (see below).
This kind of attack may be prevented by using multiple prediction algorithms, randomly chosen and changed by the server for each client.

Finally, bad network latency may cause the game world to fork into two parallel universes. If the game continues from there, it has to trust the nodes that were temporarily separated in the reconciliation process to some extent. Cheaters wait this moment to change their node parameters (player stats, etc.). If the changes are in the range of possibilities, they won't be rejected by the consistency checks. Cheaters then overload the network to separate their node, upgrade their parameters just a little bit so as to pass the consistency checks, wait for a merge to validate the changes, cause a separation again, etc. Of course, honest players end up with a very bad game experience due to both network lag and other players unfair level.
Solutions include stopping the game or rejecting a node that has become out of synchronization, but then there is the denial of service problem aforementioned. It is also possible to maintain statistics about nodes that change too often in reconciliations. Maintaining checksums of each other node parameters may help too, but is of questionable value since it precisely increases the network load at a time where it is already overloaded.

Malicious nodes

Malicious nodes run modified version of the software to gain advantage. The problem seldom appears in central server architectures, because clients usually can't take any serious decision.
In this case, a particular kind of malicious node maybe a malicious server! It may be introduced by conspiring cheaters, luring honest players to connect. This fake server would be transparent most of the time, by transmitting requests and replies to a true server in a dumb proxy setting. Cheaters can then disconnect a player and take control of her/his character. When the player next connects, for some reason, all his/her items of great values have disappeared...
As for e-business, the solution is to use a third party authentication mechanism, with server certificates signed by a trusted source (the game maker for example).

In peer-to-peer environments the solution is slightly more complicated, but very interesting. The following outline is a variant of solutions found in [18] and [11], together with a proposal to set up a trust relationship between nodes without a central server.
It consists in duplicating at least the most critical tasks to several other nodes, and keep track of inconsistencies. Example: node A, B, D are honest players, C is cheating. A decides to calculate damages done to its player and sends the parameters to B, C, and D. B and D give the correct answer A also computed, but C doesn't. Note that the effect of C cheat is void, A doesn't take it in account. But now A has serious doubts about C and announces an inconsistency to all other nodes. At this point, other nodes don't know whether A or C is wrong, or maybe none are and it was just a network lag being sync'ed out. But the point is, A and C are both tagged by B and D. If C continues to give inconsistencies and A doesn't, then it will continue to cumulate bad points. When a majority of nodes are convinced of C bad behavior, they may disconnect from it without question. Since it is clear that C will always receive correct answers, it cannot change its own values without causing an inconsistency and being automatically detected. It cannot either damage other nodes by flagging them for no reason, because doing so would flag itself too. In the end, when there are a majority a good nodes in the network, the cheating ones can't do any harm.
Trusted nodes may help, if only for keeping track of initial parameter values. Fortunately, such nodes always exist, if only to maintain connectivity, as mentioned in the Interest Management section. For added security, the trusted official game server nodes may sign data with a private key, and other nodes could use the public key for verification. Thus, there is no need for a big central server containing all critical gaming data (such a server or network may be necessary for other parameters, player account management being a good candidate).
This solution sounds very nice, but all nodes must now compute parameters for other nodes in addition to their own. Clearly, this should be reserved only for critical parameters and by splitting participating nodes in small groups (bad flags may cumulate between groups, of course).

Passive information exposure

As stated in [18], completely passive cheats may be set up to gain access to otherwise hidden information. An example presented in this article is to modify the game to use transparent walls:

it's then easy for a cheater to spot and surprise the victims. As only the cheater environment is changed, from other nodes point of view there is no give-away.
A possible solution is to use duplication and voting, like in the previous peer to peer scenario. A better solution is to note that this example supposes the server would still give the cheater information on other users even if they were hidden by a wall: this is where good interest management could be used to prevent or at least detect the cheater. This last solution is applicable to many other game types, like for “fog of war” removal detection in strategy games [18].

Other passive techniques may be much harder to detect. [17] considers the example of on-line card games: it's easy for a bridge player to set up 2 accounts and connect them simultaneously to the same game, or to contact a fellow cheater. The third player, honest, would have a very difficult time indeed.

Scripting

Another difficult to detect cheat, is to automate actions that would be too difficult or too tedious for humans. A good example is modifying the game program to automatically correct the player orientation when aiming at a target. The result is an improved hit ratio for the cheater.
This kind of cheating may be difficult to detect because from the server point of view, all the data received is plausible. The only solution known to date is to use statistical tests, and try to detect when a player is too good to be honest. Unfortunately, cheaters may also introduce a little randomness in their otherwise perfect aiming, so as to be just at the level of extremely good but honest players.
Scripting also appears in adventure games, especially massive multiplayer on-line role playing games. Automating an action for hours, like killing random monsters, improves the cheater in-game characteristics. This is much easier to detect by watchful game administrators, as such behaviors would not normally be particularly interesting for a real player.

Software bugs

Consider the following examples from [18]: A key shortcut applies to a unit it should not (Age of Empires and Starcraft), an unusual character sent in a message makes the receiver system crash when it is displayed (Firestorm), some way of executing actions faster than expected (Half-Life), a buffer overflow that executes arbitrary code on distant nodes... possibilities are endless.
While it is theoretically possible to write software without bugs, it is seldom applicable for games. The next best thing to do is to provide a very good game support after the release, and make public patches to be downloaded and distributed as often as necessary.

Conclusion on security issues

As for the other parts, this discussion presented only the main issues, and does not pretend to fully cover the subject of security in multiplayer games. For example, human factors are also important. Password attacks and social engineering techniques may be somewhat limited by educating the players, but are nonetheless a major risk. Economy can become totally unbalanced after some time in virtual worlds. Together with other cheats, and especially scripting, this leads to a real world black market of in-game content (rare items, high level avatars, etc...). The incentive for cheating is thus not going to disappear soon.
To complete the statement made at the beginning of this section, security comes with a good design, not with obscurity. Unfortunately this is not enough, and game administrators have a major role to assume after the game is released.

Summary and Conclusion

Multiplayer game development is to some extent very similar to distributed computing. The main concerns are Synchronization and scalability as the number of players increases. But games also present their own challenging problems: playability requires a very short response time, and security should be integrated in the system core.

Given so many disparate configurations [14] for the players machines, the game won't probably run at full capacity most of the time. Graceful degradation should be included as part of the design, to keep the game experience at reasonable levels even when conditions are suboptimal.

The number of players is not the only factor for game complexity, and just looking at the figures in the introduction, it can't grow much longer without exceeding the player population limit! Games will need to improve in other areas to be accepted and maintain a high quality, by using better graphics and user immersion for example, but not only.

Finally, all the problematics are inter-dependent. Security, degradation recovery, scalability and network performances, synchronization, playability, etc., no single topic can be fully optimized without impacting some of the others. This can only be done with a holistic approach to game development including all the aforementioned topics. The need for better algorithms leading to good design is more important than ever. Research is strongly needed, and much appreciated as new games come out with state-of-the-art improved algorithms. ■

Appendix: Global synchronization using a time manager.

For a LAN, and assuming the transport is reliable, a solution exist that allows both global synchronization and parallel execution of events to some extent. Many articles about distributed systems and multiplayer programming ([5], [6], [7], [10], [15], to mention a few already in reference) often talk about the now standardized IEEE 1516 HLA, it's U.S. Department of Defense ancestors (HLA 1.3, DIS), or custom implementations. The possibilities of these techniques is very vast, and certainly not restricted to the global synchronization aspect presented below. It's worth noting that relaxed conditions on participants allow for different time models, especially in HLA [19]. The goal of this appendix is not to present simulation techniques as a whole, but to give the reader a sketch of an optimized global synchronization mechanism, for completeness with the main sections.

It consists in each object using a logical (or simulated) time, instead of the real time. A time manager controls all advances in logical time at a proper pace, not necessarily 1 to 1 with real time.

All participants then agree they won't post new events in a short period after the current logical time: they can only post events for execution after a short delay. With this properly defined look-ahead period, the time manager may then order parallel execution of all events inside the look-ahead. Each participant does its jobs one by one, and notifies the time manager it is ready to advance in logical time after each job. The time manager may wait for other participants to complete their task before allowing the execution. When all events at date T are processed, the time manager can advance in logical time to the next date at which an event is scheduled. The look-ahead is useful to avoid immediate re-posts of events. This avoids blocking the time advance for other participants, and allows parallel execution of events.

The notion of logical time has the nice properties of being deterministic: the simulation can be replayed exactly the same way with the same results. It is also totally ordered (events execution time can always be compared), and global (all participants see the same ordered sequence of events). From a participant point of view, the logical time behaves exactly as the real time would.

The price to pay for all this is:
- A higher latency: messages must be exchanged constantly with the time manager.
- Some complexity in implementation.
- A dependence on each participant behaving properly for the whole system to work well. This could be a real problem if malicious nodes were introduced.

But even then, the system may work fast enough to allow for real-time execution (logical time equals wall clock time). In addition, the logical time is very handy to analyze the simulation: whether it is a step by step replay to investigate on a specific effect, or for long-term behavior if the logical time can advance faster than real-time.

These techniques are of high interest to build military and civil engineering simulations, where reproducibility and fidelity to a real system are critical, and no malicious participants are allowed because the whole system is under control. Unfortunately this is not the case in game environments, and the price is often too high to pay: games are not interested in exact fidelity, but in a good enough approximation of "reality". Yet, while a global time manager may be impractical for games, some customized version of a logical time may be very valuable at a local level.

References

1: *Multiplayer*, Wikipedia: The Free Encyclopedia, 23:51 UTC, 24 Sep 2004, http://en.wikipedia.org/wiki/Multiplayer

2: *Transmission Control Protocol*, DARPA Internet program, Request for Comments 793, September 1981, http://www.ietf.org/rfc/rfc793.txt

3: *Vulnerability Issues in TCP, Advisory 236929*, NISCC, 20 April 2004, http://www.uniras.gov.uk/niscc/docs/al-20040420-00199.html

4: *Understanding TCP Reset Attacks*, Jeremy Andrews, 2004, http://kerneltrap.org/node/view/3072

5: *A distributed architecture for multiplayer interactive applications on the Internet*, Christophe Diot, Laurent Gautier, 1999, IEEE Network, Vol. 13 Issue 4, p6, 10p

6: *A Review on Networking and Multiplayer Computer Games*, Jouni Smed, Timo Kaukoranta, Harri Hakonen, April 2002, Turku Centre for Computer Science, http://citeseer.ist.psu.edu/article/smed02review.html

7: *Cheat-proof playout for centralized and distributed online games*, Nathaniel E. Baughman and Brian Neil Levine, April 2001, Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference, http://citeseer.ist.psu.edu/baughman01cheatproof.html

8: *Scalability With a Big 'S'*, Crosby Fitch, February 26, 2001, http://www.gamasutra.com/features/20010226/fitch_pfv.htm

9: *A Federated Peer-to-Peer Network Game Architecture*, Sean Rooney, Daniel Bauer, Rudy Deydier, May 2004, IEEE Communications Magazine, Vol. 42 Issue 5, p114, 9p

10: *Interest Management in Large-Scale Virtual Environments*, Katherine L. Morse, Lubomir Bic, Michael Dillencourt, February 2000, Presence: Teleoperators & Virtual Environments, Vol. 9 Issue 1, p52

11: *Peer-to-Peer Support for Massively Multiplayer Games*, Björn Knutsson, Honghui Lu, Wei Xu, Bryan Hopkins, March 2004, Proceedings of the Twenty-third IEEE Computer and Communication Society INFOCOM Conference

12: *Efficient representation and streaming of 3D scenes*, J. Sahm, I. Soetebier, H. Birthelmer, February 2004, Computers & Graphics, Vol. 28 Issue 1, p15, 10p

13: *Defeating Lag With Cubic Splines*, Nicholas Van Caldwell, 2000, http://www.gamedev.net/reference/articles/article914.asp

14: *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*, Yahn W. Bernier, February 2001, Game Developers Conference, http://www.gdconf.com/archives/2001/bernier.doc

15: *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*, Sandeep Kishan Singhal, August 1996, Stanford University, http://www.dsg.stanford.edu/singhal/thesis.ps

16: *Statistical Client Prediction*, Jakob Ramskov, 1999, http://www.gamedev.net/reference/articles/article876.asp

17: *Security issues in online games*, J.J. Yan, H.J. Choi, November 2001, Proceedings of International Conference on application and development of computer games in the 21st century, pages 143-150

18: *How to hurt the hackers: The scoop on Internet cheating and how you can combat it*, Matt Pritchard, 24 July 2000, http://gamasutra.com/features/20000724/pritchard_pfv.htm

19: *The department of defense high level architecture*, J. S. Dahmann, R. M. Fujimoto, R. M. Weatherly, December 1997, Proceedings of the 1997 Winter Simulation Conference, pp. 142-149, http://citeseer.ist.psu.edu/dahmann97department.html