pSense - Maintaining a dynamic localized peer-to-peer structure for position based multicast in games*

Arne Schmieg
Techn. Univ. Darmstadt
Germany
a.schmieg@dvs.tu-darmstadt.de

Michael Stieler
Techn. Univ. Darmstadt
Germany
m.stieler@dvs.tu-darmstadt.de

Sebastian Jeckel
Techn. Univ. Darmstadt
Germany
s.jeckel@dvs.tu-darmstadt.de

Patric Kabus
Techn. Univ. Darmstadt
Germany
pkabus@dvs.tu-darmstadt.de

Bettina Kemme†
McGill Univ., Montreal
Canada
kemme@cs.mcgill.ca

Alejandro Buchmann
Techn. Univ. Darmstadt
Germany
buchmann@dvs.tu-darmstadt.de

Abstract

This paper presents an algorithm for creating and maintaining a dynamic localized peer-to-peer overlay network with its main application to massively multiplayer games. In these games, players reside in a large game world with many thousands of players but each player has typically a limited vision range. In our solution, players join the network as peers and mainly connect to neighbor peers that are close to them in the virtual game world. As players move in the game they change their neighbors dynamically with very little overhead. Peers can multicast messages that are received by peers in their locality very fast (often faster than in client-server solutions) while players that are further away receive them later or not at all. Not receiving messages from remote players is important in order to not cause the load on each peer to grow with the number of players in the game. Our performance analysis confirms that our solution allows for dynamic game worlds of practically unlimited size, only limited in scale by the number of players within the vision range.

1. Introduction

Massively multiplayer online games (MMOG) allow thousands of concurrent users to play together in a persistent game world. They are typically run in a client/server architecture. The client software renders the game world. Each client is typically able to control one player. Players can perform actions such as updating their position in the game world, interacting with objects (e.g., picking up an object), or interacting with other players. Each player sends its actions to the server. The server serializes them and then sends each valid action to those players that are interested in this action. Typically, each player has only a limited vision range seeing only the part of the world around its current position. While this is typically part of the game semantics, it is also often a technical requirement. The relatively weak machines on which the client software runs are sometimes not able to receive and process actions of all players in the game. Therefore, the server has an interest management module [19, 4] which determines for each player and action, whether the action is relevant for the player, and only information about those actions is transferred to the player.

But the sever side also faces scalability problems. First, the server can only connect to a limited amount of clients and propagate a limited amount of information across these links. Second, the server has only limited processing capacity. For each action, the interest management module must determine the interested players. Several action types require some serialization and conflict detection (e.g., not two players may pick up the same object). Thus, a standard server can typically only support a few thousand players. Beyond that, the most common approach by commercial providers is to use a multi-machine server cluster which adds a lot of complexity in order to achieve load-balancing and borderless player experience [6, 1].

As an alternative, proposals have been made to run MMOGs over a Peer-to-Peer (P2P) infrastructure (e.g., [16, 2, 13, 11]). A major issue is the implementation of in-

*This research was partially funded by Deutsche Forschungsgemeinschaft through the Research Training Group "Enabling Technologies for e- Commerce", the Research Group "QuaP2P", and Activator.

terest management without having global knowledge. Current approaches typically split the game world into small zones. A master node (peer) is the server for players in this zone controlling actions that require serialization [16]. For the dissemination of position updates to all players of a zone, direct multicast can be used [16] or the master node is also in charge of disseminating position updates [13]. Having predefined zones as units of distribution has several problems. First, mobility is not well supported. When a player moves from between zones, an expensive hand-shake has to connect a player to a new server. If master nodes leave the game, an expensive reconfiguration is necessary. Second, zones build visibility boundaries. Players cannot see players of other zones even if they are close to these zones. This might be acceptable in game worlds where buildings and rooms build natural area boundaries but not in larger game spaces. Finally, load-balancing is difficult if the number of players in a zone changes dynamically. However, only few allow for dynamically changing the size of zones, and if they do, it is cumbersome [12, 10].

In this paper, we present a P2P solution that is specifically designed to handle interest management and support dynamic behavior. Our focus currently is on the efficient multicast of position updates as they make up by far the largest part of all actions, thus have the biggest effect on performance and scalability. We consider serialization of potentially conflicting actions as an orthogonal topic. It could be handled, e.g. by a central server, because these types of events are much less frequent then position updates. The separation of these two concerns has been very useful, as it allows for a much better optimization of the problem at hand. Our solution is completely decentralized where each node has exactly the same responsibilities. It fulfills three fundamental requirements: (i) Players in the neighborhood get updates very soon (as good or even better than in a client/server approach). (ii) Each node only needs a limited amount of bandwidth; this automatically means that remote players should not get too many irrelevant messages since they would overflow their machines. (iii) Players can move freely in the game world with the set of players that are visible to them changing in a continuous manner.

The main ideas are as follows. We maintain an overlay in which each node keeps information about *neighbor players*, i.e., players in the vision range of the local player, as accurate as possible. In order to be able to update its neighbor view fast if movements occur, and also to avoid network partitions, each node keeps track of some remote players. We call our approach *pSense* as nodes continuously sense the positions of players and detect whether new players enter or leave the vision range. A *localized mulitcast* sends position updates only to nodes with neighboring players, and this with very little delay (often only one or two hops). We exploit the fact that position updates don't really need to be delivered reliably. If a player has not received a specific position update after one or two rounds, there is no use to deliver it but to deliver the next update as soon as possible. Position update messages are not only used by the application to update the game world but also used to maintain the peer overlay. Overall, each node only maintains a limited amount of information, and sends a limited number of messages per time unit, as individual nodes have storage, bandwidth and processing constraints.

pSense has been developed with MMOGs in mind. Nevertheless, we believe that it can be used by other applications. For example when rescue teams are deployed in large disaster areas and each member is equipped with a handheld communication device it helps individuals to keep track of nearby team members, and thus, help to coordinate the rescue effort. In this situation, the vision range would not reflect the proximity in a virtual but in the real world.

In summary, our P2P architecture

- provides a straightforward, localized multicast that is sent only to nodes in the neighborhood;
- maintains an overlay using probabilistic measures in order to handle the dynamism of the system.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 describes the pSense algorithm. Section 4 presents our implementation and our metric for the "goodness" of a localized multicast. Section 5 discusses our performance results. Finally,Section 6 concludes and discusses future work.

2. Related Work

Probably the most well known P2P MMOG approach is SimMud [16]. Peers are member of a Pastry [23] DHT overlay. The game world is distributed in zones and a peer is server (master) node of those zones and objects whose identifiers are closest to the peer's own identifier. Game state is disseminated using the Scribe multicast [5]. As Scribe is tree-based, this leads to delays of several hops. An improvement over SimMud was proposed in [13] where most of the traffic is handled via the master node leading to an average latency of two hops. In MOPAR [24], the master node of a zone maintains for each player p a neighborhood list with all the players in p's vision range. As MOPAR allows the vision range to overlap with several zones, masters of neighboring zones have to exchange player information. Players communicate directly with each other based on the neighborhood lists allowing for fast multicast of position updates. However, the maintenance of neighborhood lists, and the movement of players from one zone to another is complex. In Mercury [2], concepts of DTH-based content publish/subscribe are used to enable delivery of events to only a subset of players. As mentioned before, zone-based

approaches have many disadvantages. Our approach has no predefined zones but uses one single continuous space.

Most similar to our work are [12, 14, 15]. All three propose a decentralized P2P approach where players keep track of other players in their continuously changing neighborhood. However, connectivity of the entire network can easily be lost [14] or requires the maintenance of complex and process intensive structures such as Voronoi diagrams [12] or convex hulls [15]. In [14, 15] only overlay maintenance is considered, while [12] is similar to us in that it exploits multicast for overlay maintenance. In [14], the neighborhood always consists of a fixed number of players determined by the bandwidth capacity. [15] handles processing limitations by decreasing the vision range. In contrast, our approach supports all neighbors in the vision range as defined by the application. If bandwidth is limited, performance will become worse, but as our experiments show, only to an acceptable degree. None of the three approaches provides overhead or performance evaluations.

Commercial systems rely heavily on the client-server architecture. P2P systems are only provided in very small scale, e.g., Z-Net supports up to 32 players [22]. All players interact directly with each other. The number of players reflects the bandwidth limitations of current machines connected to the Internet. In our approach, we can support a similar number of players within the vision range with nearly optimal performance. Additionally, we allow the players in the vision range to continuously change, we support more than 32 players in the vision range with only slightly decreased performance, and we support an unlimited total number of players in the entire game world.

Many P2P multicast solutions build a dissemination tree [17, 5] in order to distribute the message propagation across all nodes. This makes only sense if not all nodes are senders. If some nodes in the tree are not interested in a message themselves, they receive it unnecessarily. In general, tree-based multicast has been developed for applications with a large receiver base. However, in MMOGs a message is typically only of interest for few players. Group communication systems (GCS) [7] offer primitives to multicast messages within a group of sites. However, joining or leaving a group are typically expensive operations. If GCS would be used in MMOG there would be still the task to determine when players have to join which groups.

There exist many proposals for probabilistic multicast. A message could be first multicast via a tree, while gossiping among neighbor nodes is used to recover lost messages [3]. Another option is to directly multicast a message via gossiping [9]. This resembles the combined flooding/random walk searches in unstructured P2P systems [18]. Each message is multicast to a subset of neighbors which in turn forward it to a subset of their neighbors until a certain depth is reached. Such approaches are not suited for several reasons. Firstly, gossiping aims at achieving high reliability by introducing a lot of redundancy. Nodes will often receive a message several times. Given the amount of messages in a MMOG, nodes are likely to be too overloaded with such an approach. In fact, our approach goes into the opposite direction. If the bandwidth is too small, a node will actually not receive all messages of a neighbor player but only enough to have a good game experience. Secondly, these gossip based protocols aim at propagating messages of all nodes to all other nodes in the system. In contrast, in a MMOG, each message should only reach a small number of players.

3. Algorithm

In this section we describe the algorithm behind pSense. For the discussion we take the view of an arbitrary node in the network which we identify as the *local node*. Every node hosts a player in the game. In the following, we use the words peer, player and node interchangeably. Every time we talk about the position of a node and its distance to other nodes, this refers to the position of the player within the game world and its distance to other players.

3.1. Main Concepts

pSense has to handle two main tasks. Firstly, when the local player moves, a position update should be sent to those players that are interested in the movement. Interest is determined by a player's limited vision range. The vision range delimits the area in which a player can perceive changes of the game world[1]. Changes outside this area are not of interest to the player. Players in the vision range of the local player are called the *neighbors* of the local player. We assume the neighbor relation to be symmetric. Thus, in order to achieve a good game performance, the local node needs to send as fast as possible position updates to its neighbor players since they are the ones that have the local node in their vision range. Other players should not receive the update in order to not overburden them.

The second task is to keep the player network connected. Restricting message exchange to only the neighbor nodes in the vision range bears a high risk for generating network partitions. Often players tend to gather in certain locations. If these locations are far apart from each other, the nodes of one location could completely lose contact to the nodes of other locations. Since we do not have any superpeers with global knowledge, these partitions cannot be joined again. Last but not least, noder are not static. Each node constantly changes its position whenever the player walks around. Thus, a node has to detect fast when other players

[1] In this paper we are talking about player objects but in principle we could address any changes of objects within the game world.

enter or leave its vision range. This means, pSense has to function in a highly dynamic environment.

Overlay Maintenance In order to restrict updates mainly to nodes in the vicinity while at the same time avoiding network partitions, every node of a pSense network maintains two lists: a list of near nodes and a list of sensor nodes.

The *near node list* contains only neighbor nodes, that is, peers that are within the vision range of the local node and that need position updates very fast. The local node attempts to keep its list as accurate as possible. As the local node detects new neighbors it adds them to its near node list. Nodes that have left are removed. But of course, changes in the neighbor configuration cannot be detected instantaneously, and thus, the list might miss some neighbor nodes while containing some nodes that are not in the vision range. We refer to nodes in the near node list as *near nodes*.

The *sensor node list* contains nodes that are just a bit outside the vision range and stick out like antennas in every direction. The purpose of the sensor node list is to avoid network partitions by keeping contact to more distant nodes and to detect nodes that move closer to the local node. If a sensor node detects an approaching new node it can introduce this node to the local node. Sensor nodes should be distributed as evenly as possible around the local node to provide the best chances of keeping connections to the rest of the network and to detect new approaching nodes.

Figure 1(a) shows an example of how the network is seen by a local node. All nodes that are known to the local node and within the vision range are marked as near node. The sensor nodes are peers just outside the vision range distributed around the local node as evenly as possible. Typically, the local node knows most of its neighbors.

Localized Multicast In pSense, when the local node changes its position it sends a position update message directly to its near nodes and sensor nodes. If the number of nodes in these two sets exceeds the outbound (upload) bandwidth capacity of the local node, a random subset is chosen as destination. Each near node that receives this message has the accurate position of the local node within one hop. As this original message might not have reached all neighbor nodes, some forwarding is performed. When a neighbor or sensor node of the local node receives the update, it forwards it to those nodes it knows that reside in the vision range of the local node. A sensor node might not know such node. In this case, it simply tries to forward the message closer to the local node. While forwarding position updates we try to avoid sending duplicates to keep bandwidth consumption low. Messages are also discarded after few forwards. Forwards are mainly needed to detect new neighbor nodes fast. They also provide reliability. However, one has to be aware that receiving a relatively old position update has no benefit if a fresher update has already arrived.

Adjusting Sensor Nodes Additionally to position updates, the local node sends *sensor node request* messages to its current sensor nodes. A sensor nodes answers with a *sensor node suggestion* message that contains the identifier and the position of the node it thinks is the most appropriate sensor node. This could be itself or another node.

3.2. Implementation

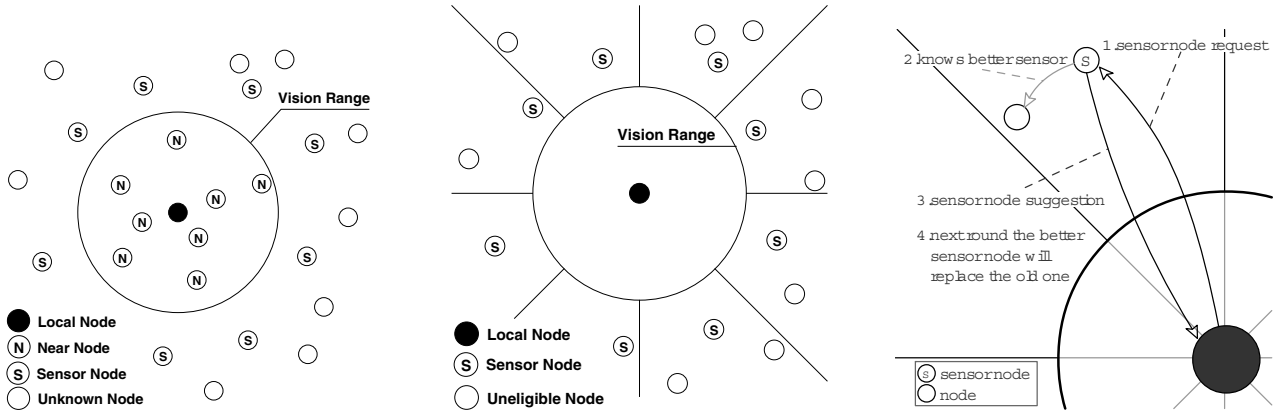
We now give a step-by-step description of the pSense algorithm. The local node performs the following steps.

1. Receive Messages This step is performed whenever the local node receives a new message. Such messages can arrive at any time. A message could be a position update message, or a sensor related message. First the hash of the message is compared to a list of seen hashes to avoid processing duplicates. Any duplicates are immediately discarded. Additionally, if the message is a position update and older than a previously received position update of the same originator it is discarded. For that purpose, position updates are tagged with node specific sequence numbers. A current position update is then delivered to the gaming application. Finally, all received messages that are not discarded are put into an incoming message queue. This incoming message queue is then further processed in Step 2.

2. Round-based Overlay Maintenance and Mulitcast The main actions at the local node are performed periodically. This resembles current games where the server forwards game state changes in rounds to the players. Each round performs the following steps:

a. Update Lists In this step, the local node updates the near node and sensor node lists. In order to do so, all position updates and sensor suggestion messages that have been enqueued in the incoming message queue are checked whether they contain updated positions for already known nodes or positions of previously unknown nodes. All nodes that are within the vision range are put into the near node list. From the remaining nodes, the best candidates are chosen as sensor nodes (see Section 3.3 for details). Nodes that are neither a near node nor a sensor node are discarded.

b. Determine Outgoing Messages In this step, we determine messages that the local node wants to send. In the following, when we talk about sending a message, we actually mean putting it into an outgoing message queue.
(i) First, the current position of the local player is determined. A position update message is sent to each node in the near node list and the sensor node list. Each of these position update messages also contains a list with the identifiers of *all* nodes in the near node list. The list of these receiving nodes is called the *receiver list*. This helps to reduce the number of duplicate messages as discussed below.



(a) Local View of the Network (b) Sensor Node Selection (c) Sensor Node Maintenance

Figure 1. pSense Algorithm

(ii) To each sensor node a sensor request message is sent in order to detect the most accurate sensor nodes.
(iii) Then, the messages in the incoming message queue are processed: First, for every sensor request in the incoming message queue a sensor node suggestion is created (see Section 3.3) and sent back to the originator. Then, each position update message in the queue is potentially forwarded if it has not yet reached its life-time limit. Life-time is measured in the number of hops it has travelled so far (i.e., one plus the number of forwards). For each non-expired update message, the local node checks which of the nodes in its near node and sensor node lists are neighbors of the originator of the message but are not contained in the receiver list of the message. To all of these nodes the update is forwarded directly because they have potentially not yet received the message. The receiver list is adjusted accordingly to also include these new receivers. If the local node has received this position update because it is the sensor node of the originator of the message, it might not know any neighbor node of the originator. In this case, the message gets forwarded to the node which is closest to the originator.

c. Send Messages Before sending any message, if the amount of messages in the outgoing message queue exceeds the upload bandwidth of the local node, the local node deletes random position updates from the queue until the amount of traffic fits. The receiver list of the remaining updates is shrunk appropriately. Sensor node requests and suggestions are never deleted from the queue. Finally, the local node sends the remaining messages to its recipients and purges both queues.

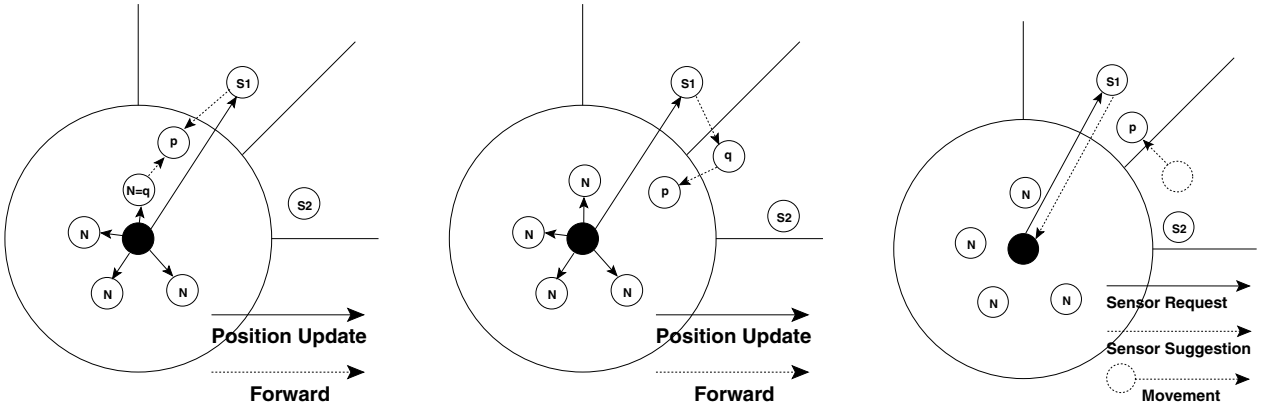
3.3. Selecting Sensor Nodes

As described above, the sensor nodes are just outside of the vision range of the local node and should be evenly distributed around it. In our solution for a 2-dimensional game world, we draw a circle around the local node and partition it into sectors of equal size. For each sector, the closest node that is outside the vision range is chosen as sensor node. Figure 1(b) shows an example of this selection. While other solutions are possible, our solution has the advantage of being simple and at the same time our experiments with both highly and very sparsely populated game worlds have shown that nodes are always detected very fast.

Of course a sensor node can only be selected among the nodes that are known to the local node. Since the local node usually doesn't receive updates from outside its vision range, it can hardly find better sensor nodes on its own. For this reason, the local node asks each of its current sensor nodes periodically whether they know a better candidate for this sector. If there is no sensor node for a certain sector, the local node sends the sensor request message to a node from the near node list which is part of this sector or any other node it knows that is close to this sector. This sensor request contains the current position of the local node and the sector identifier. The sensor node has a good knowledge of its vicinity and checks whether there is a node that is better suited than itself. It then sends the identifier and the position of the node it has chosen (it could be itself). The local node then replaces the old sensor node with the new one. This process is shown in Figure 1(c).

3.4. Joining and Leaving the Network

As players usually may start and stop playing a game at any time, pSense must be able to handle arbitrary joining and leaving of nodes. In order to join the network, a new node must only know a single random node which is already part of the network. If there is a central server, for example for authenticated login, the central server could provide the address of such a node, which we refer to as *old* node.



(a) Localized Multicast (b) Indirect Forwarding (c) Sensor Node Adjustment

Figure 2. Examples

If the old node is within vision range, the new node sends directly updates to it. The old node adds the new node to its near node list and starts forwarding position updates to the new node. Thus, the new node soon gets to know other nodes in its vision range. If the old node is not within vision range, the new node will send it a sensor node request. From the nodes it knows, the old node suggests a better sensor node than itself to the new node. The process repeats, and the new node will successively find better sensor nodes. Additionally, the new node sends position updates to its current sensor nodes. They try to forward them to nodes which are closer to the new node. Eventually the updates will reach a node within the vision range of the new node. From then on, the new node soon receives updates from nearby nodes and populates its own near node list.

When joining, a node might also need to receive the map of the game world, information about non-player objects, etc. How this is handled is orthogonal to this paper.

When a node leaves the network, it does not need to perform special operations. The remaining nodes may lose at most either a sensor node or a near node. In case they lose a near node, they simply stop sending messages to it. If they lose a sensor node, they choose a new one as described in Section 3.3. If nodes leave, there are very few situations where the network could become disconnected. For instance, when all players are in one straight line. In our experiments, however, such situation never occurred. This problem could be solved with backup sensor nodes.

The system automatically bootstrap when the first node joins the network. If this is done, as described above, over a server, this server can tell the node that it is the first.

3.5. Examples

We present now several examples that provide some intuitive reasoning for the individual steps of the pSense algorithm. In Figure 2(a), we see the local node, its vision range and nodes in its surrounding. We only show two sectors outside the vision range with its sensor nodes. Let's assume that the local node and peer node p, although in each others vision range, don't know each other because p only moved recently into this area. When the local node multicasts its next position update it sends it to all nodes in the vision range except of p. However, as node q is close to p it is likely that q already knows p. Thus, it forwards the position update to p. Peer node p registers the local node in its near node list. When p sends its next position update, the local node will receive it, put p in its own near node list and have accurate information about p. If no node q exists in the vision range of the local node that knows p, then there is still the possibility that the sensor node $s1$ knows about p. As it also receives the position update of the local node, it forwards it to p and the two nodes get to know each other.

Figure 2(b) shows how position updates are forwarded via sensor nodes. In the example, no sensor or near node knows p but a node q close to p outside the vision range is known by sensor node $s1$. When $s1$ receives the position update of the local node, it forwards it to q as q is closer to the local node. Node q, in turn, forwards the update to p.

Finally, Figure 2(c) shows an example of how a new sensor node could be determined. In the example, p has moved very recently into the sector for which $s1$ is the sensor node. At this time point p is now the preferred sensor node for this sector. As p is in the vision range of $s1$, $s1$ knows p or will soon get to know it. When the local node sends its periodic sensor node request message to $s1$, $s1$ determines that p is now a better sensor node and responds to the local node with a sensor suggestion message containing p and its position. Note that before the move of p to the different sector, p was of no interest to the local node, because in its original sector, the sensor node $s2$ was closer than p.

4. Simulation Environment

We implemented the algorithm using the PeerSim simulator [20]. PeerSim provides a round-based modus where in each round a node can send messages to other nodes, receive messages, and do some local processing. In our implementation, a message is received one round after it has been sent, and this counts as one message hop. The software running on each node is split into two modules. The first is the overlay network implementing the algorithm described in the previous section. The second is an application that represents one player and the corresponding game client.

4.1. Simulated Game

In each round of the simulation, the game application on a peer moves its player and then multicasts the new position via the overlay network. Furthermore, it processes all position updates delivered by the overlay network. As discussed before, the position of a peer and its player are conceptually the same so that we denote the player on peer p also with p.

Initially, player p does not know the position of any other player. The first position update it receives for a player q determines the initial position of q in the game world seen by p. This position of q is updated whenever a newer position update from q arrives.

Our simulator has two modes for player movement. In *random mode*, each player moves with discrete steps into a random direction. Each round the direction can change with a certain probability. This mode spreads players equally across the game world. Using the *hot spot mode*, the game world has a certain number of places where players tend to gather. They first move to a hotspot they are attracted to. Then they perform random moves within a certain range of the hot spot. After a random time interval they choose a different hotspot and move to it. Thus, there are many players in each hot spot area and on the paths connecting two hotspots while the rest of the game world is mainly empty. This mode resembles the distribution of players in certain games more closely than the random mode.

4.2. Measuring quality

We consider as the protocol quality the freshness of the information each player has about other players in the system. For players p and q we express p's knowledge of q with the age of the last position update p received from q. We calculate $PositionAge(p, q)$ as the difference between the current round in the simulation and the round at which q initiated this position update. In perfect circumstances, p receives the update one round after q sent it, in which case $PositionAge(p, q) = 1$. If p has not yet received any position update from q, then $PositionAge$ is set to a fixed maximum value. In our experiments, we used a maximum of 20. In general this will be an application dependent value.

Our metric of freshness does not directly take $PositionAge$. Instead, it distinguishes two areas within the vision range. Around a player p there is a small interaction range containing players with whom p can potentially directly interact (e.g., talk or fight). The information about a player q in this interaction range needs to be fresh. Thus, for player q in the interaction range of p we take

$$PQ(p, q) = PositionAge(p, q)$$

For a player q in the vision range that is not in the interaction range, the importance of the accuracy of the information decreases with increasing distance between p and q. That means, the further q is from p the less should a large value for the position age have a negative impact on the protocol quality. Let IR denote the radius of the interaction range, VR the radius of the vision range, and $dist(p, q)$ the distance between p and q. Then

$$PQ(p, q) = PositionAge(p, q)^{\left(1-\frac{dist(p,q)-IR}{VR-IR}\right)}$$

For instance, given $PositionAge(p, q) = 3$, $IR = 2$, $VR = 5$ and $dist(p, q) = 3$, then $PQ(p, q) = 2.08$.

Finally, players that are outside p's vision range have no influence on the protocol quality. With this, given a round in the simulation, the protocol quality for player p with players $q_1, q_2, ...q_m$ in its interaction and vision range is

$$PQ(p) = \frac{1}{m}\sum_{i=1}^{m} PQ(p, q_i)$$

The overall protocol quality PQ for a given simulation round is the average over all $PQ(p_i)$, p_i being a player in the game. Finally, $\overline{PQ}$ is the average over the PQ values of all rounds within a simulation run. $PQ = \overline{PQ} = 1$ if all players send all position updates directly to their neighbors.

5. Evaluation

We have conducted a wide set of experiments. Table 5 describes the main parameters and the standard settings if not indicated otherwise. We have chosen a relatively large vision range given the overall game size. The reason is that network size was restricted by the runtime of the simulation and we tested only up to 600 nodes. However, as we have mentioned before, performance mainly depends on the number of players in the vision range which we chose large. For the bandwidth capacity, we found information on the Web indicating that around 20% of Internet sites have an upstream (outbound) bandwidth capacity of 128 KBit/s, another 20% have 256 KBit/s and the rest have larger capacity. Downstream (inbound) capacity is usually larger. One can

Parameter	Value
Standard Game Size	1000×1000
Vision Radius	200
Interaction Radius	50
Outbound Bandwidth Cap per Round	5KByte
Simulation Rounds	500

expect that these bandwidth limits will increase fast in the near future. For a good game experience, each peer should run at least 3 multicast rounds per second. Thus, our standard bandwidth cap is at 5 KByte per round (43 KBit) so that even the weakest nodes are able to multicast 3x per second. All tests were run over 500 simulation rounds.

5.1. Scalability

We claimed that performance does not depend on the overall number of players but only on the number of players in the vision range. We first want to confirm this claim.

Fig. 3(a) shows the PQ for each of the first 500 rounds using the Random game type. One run has 100 players and the standard game size of 1000x1000. A second has 300 players with the same game size. The third run has 300 players on 3-times the standard game size. We observe that the quality shows very little variation throughout the experiment except. The PQ with 100 players and standard game size, and 300 players with 3x the game size are identical as the number of players in the vision range are the same (around 8). In both cases, the PQ is very close to 1, the optimal value. This means, most players in the vision range are informed in one hop. With 300 players and standard game size, the PQ value is slightly worse. There are too many players in the vision range (around 25) and not all receive each position update in one hop. Instead, some receive it in two or more hops or might not receive it at all as a more current one is already in circulation. Nevertheless, the PQ is still at a very low 1.15.

We also looked at the 90-percentile PQ value meaning that 90% of players perceived a performance equal or better to this value. This provides a good indicator of the variance seen in performance. Only for 300 players on the standard game size did the 90-percentile value differ significantly from the average and is shown in the figure. It is at around 1.3, 10% worse than the average. This shows that there is little variation and players get informed very quickly about new players in their vision range.

Our second scalability test analyzes the performance for both the Random game type and the Hotspot game type with 10 hotspots. It also compares against a client/server system. For a client/server system, we assume that each round each player p sends its position update to the server which forwards it the next round to the players in p's vision range. Thus, the position age for players in the vision range is always 2. However, PQ is smaller than 2 as the formula for PQ weights the position age with the distance to the player and if $PositionAge(p, q) = 2$, then $PQ(p, q) < 2$ for a player q outside p's interaction radius. We assume no bandwidth limitation at the server.

Figure 3(b) compares $\overline{PQ}$ of pSense with the client/server solution for the game types Random (indicated with R in the figure) and Hotspot (HS in the figure). Each block of bars shows first the results for pSense and Random, then for client/server and Random, then for pSense and Hotspot, and finally for client/server and Hotspot. In each block, the total number of players was selected, so that independently of the game type around the same number of players are in the vision range. The x-axis shows the number of players in the vision range, the y-axis the $\overline{PQ}$ value.

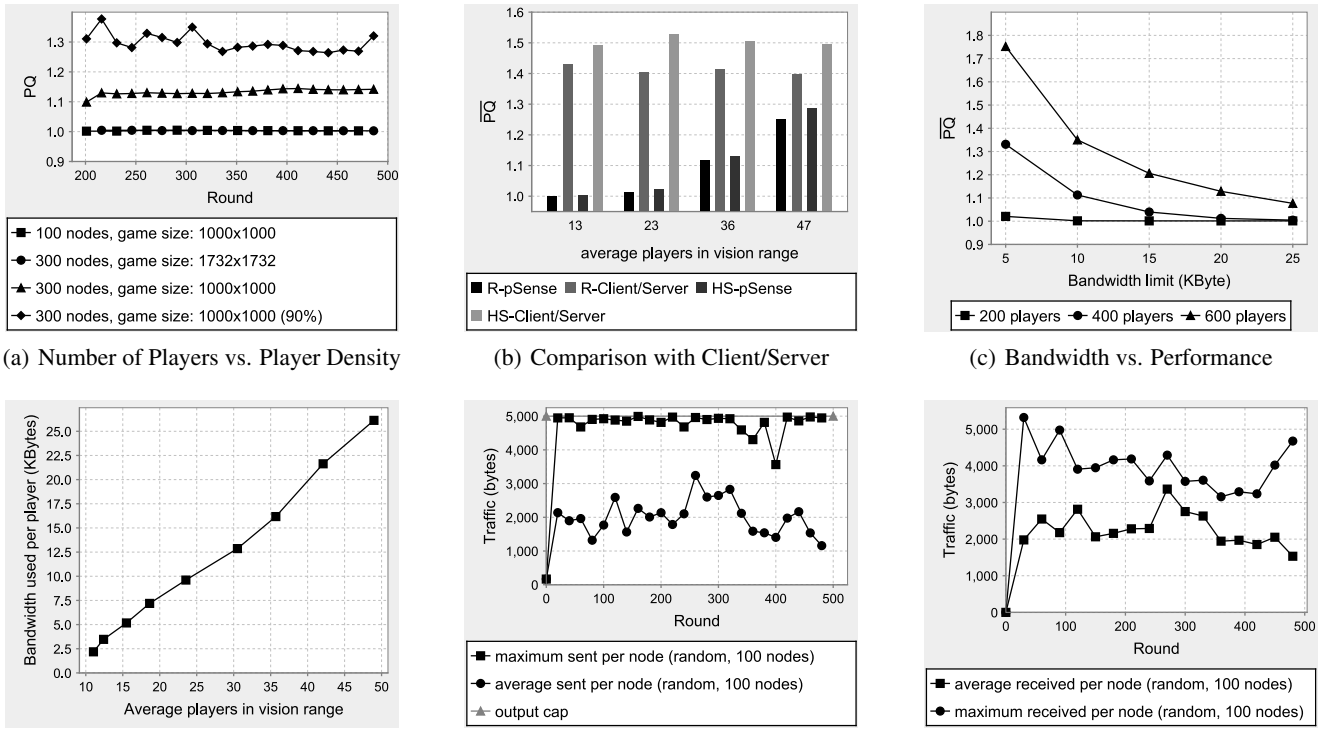
Our first observation is that pSense always has significant better performance than the client/server architecture. As most messages are sent with one hop latency, $\overline{PQ}$ is very close to 1 as long as the bandwidth is not the limiting factor. But even with nearly 50 players in the vision range, it has much better performance than the client/server solution, although forwarding and message drops occur. In contrast, the client/server architecture has a $\overline{PQ}$ of around 1.4-1.5.

The $\overline{PQ}$ of the client/server system depends on the game type and not the number of players in the vision range. As we do not consider any processing or bandwidth limitations at the server, the $\overline{PQ}$ value is only determined by the 2-hop latency for all messages. Hotspot has a worse $\overline{PQ}$ than Random. This could be caused by Hotspot having more players in the interaction range than Random. The latency of these players has a large influence on the PQ value.

For pSense, performance decreases with increasing number of players in the vision range. When the capacity limit kicks in, some players receive messages only through relaying or not at all. In this figure one can nicely see that a low-bandwidth node is able to directly broadcast to approximately 30 nodes – the number that is supported by commercial P2P gaming infrastructures [22]. While the neighborhood of a player continuously changes, the player learns quickly about new players in its vicinity through its near and sensor nodes. Although the changes can be very extreme in the case of Hotspot, performance remains excellent.

5.2. Bandwidth vs. Performance Trade-offs

If a player could always send all position updates directly to all its neighbors, then PQ would always be 1. However, due to bandwidth limitations this might not always be possible, and only a subset is selected. Thus, nodes receive a position update only after 2 or more hops or not at all, negatively affecting PQ. In this section, we analyze how the bandwidth capacity influences the PQ value.



(a) Number of Players vs. Player Density (b) Comparison with Client/Server (c) Bandwidth vs. Performance

(d) Bandwidth Requirement (e) Outbound Bandwidth over Time (f) Input Bandwidth over Time

Figure 3. Performance Results

We run the Random game type with 200, 400 and 600 players (respectively leading to approximately 16, 35 and 65 players in the vision range). Figure 3(c) shows the $\overline{PQ}$ when we increase the outbound bandwidth capacity from 5 KByte to 25 KByte per round. We estimate that currently around 20% of nodes can support 25 KBytes per round.

We can observe that a game with 200 players has already excellent performance at a bandwidth of 5 KByte per round while the performance considerably suffers for more players. With 600 players, the performance at 5 KByte is worse than in a client/server system. However, as more bandwidth is available, performance quickly improves and already with 10 KBytes, the $\overline{PQ}$ is below 1.4 for 600 players. After that, adding more bandwidth capacity further improves the performance until in reaches 1.

Our second experiment shows how much bandwidth is actually needed to achieve $PQ = 1$. Fig. 3(d) shows for the Random game type for different number of players in the vision range how much bandwidth was used on average when no bandwidth cap was used. We can see that bandwidth requirement increases linearly with the number of neighbors.

In our current implementation, node ids take up 4 Bytes and no compression is used. We believe that performance could be further improved if more engineering work would be put into marshalling position messages.

5.3. Traffic over Time

Finally, we have a closer look at the traffic behavior. Figure 3(e) shows the average and maximum outbound traffic in each of the first 500 rounds for a random game with 100 players and 5 KByte bandwidth cap. In this experiment the average PQ is close to 1. We can see that on average, the network traffic is well below the maximum. However, in each round there are some nodes who have reached the maximum value and putting a cap is important to not temporarily overload these nodes. Figure 3(f) shows the inbound traffic. Here, the averages are similar to the outbound The maximum values are well controlled by putting a cap on the maximum value for outbound messages.

5.4. Summary

Our approach provides excellent performance for large multiplayer games with continuous game worlds, outperforming the client/server architecture as long as each player has only a limited number of neighbors.

6. Conclusion and Future Work

pSense is a P2P solution that combines overlay maintenance with a localized multicast that only sends messages

to peers in the neighborhood. Peers can change their location in the overlay dynamically so that they are close to their neighbors from an application point of view. Dynamism is handled by keeping track of some remote nodes which allow for a fast detection of approaching nodes. The localized multicast then sends messages to the closest nodes very efficiently while most remote nodes don't receive the messages at all. The multicast is indirectly used to determine the relative distance of peers and thus, rearrange the overlay dynamically. pSense is completely decentralized and allows to freely scale in terms of network size.

pSense was developed in the context of massively multiplayer games. It presents a novel P2P gaming infrastructure which allows for a very fast dissemination of position updates. Position updates are the most frequent action type players perform in the game world. Thus, their efficient propagation to interested parties is very important.

There is plenty of interesting work that is left to be done. Tests on real game traces could give further insight into the performance of the system. Furthermore, the effect of heterogeneous peer setup has not yet been well analyzed. An initial experiment, where 50% of nodes had a low bandwidth capacity and 50% had a high capacity, achieved a $\overline{PQ}$ that was exactly between the $\overline{PQ}$ values for all low-capacity nodes resp. all high-capacity nodes. The protocol could be adjusted to better exploit capacities in a heterogeneous environment. The choice of sensor nodes has shown to work well. But we are not quite sure about the appropriate number of sensor nodes and their distance. We have also not yet considered cheating. One would have to understand the possibilities of cheating [8, 21] in this context and then see, how cheating could be addressed so that it does not have an affect on the performance. Finally, position updates are only one type of action performed in games. Other action types, if they should also be handled in a peer-to-peer fashion, will need some serialization and conflict detection, and thus, require a more sophisticated multicast.

References

[1] R. K. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. In *Int. Middleware Conf.*, pages 390–400, 2005.

[2] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.

[3] K. P. Birman, M. Hayden, Ö. Özkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[4] J.-S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Int. ACM Workshop on Network and System Support for Games (NETGAMES)*, 2006.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), Oct. 2002.

[6] J. Chen, B. Wu, M. DeLap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *ACM Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[7] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[8] S. B. Davis. Why cheating matters - cheating, game security, and the future of global on-line gaming business. In *Game Developers Conference*, 2003.

[9] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[10] C. GauthierDickey, V. M. Lo, and D. Zappala. Using n-trees for scalable event ordering in peer-to-peer games. In *Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2005.

[11] C. GauthierDickey, D. Zappala, V. M. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV*, pages 134–139, 2004.

[12] S.-Y. Hu and G.-M. Liao. Scalable peer-to-peer networked virtual environment. In *NETGAMES*, 2004.

[13] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NETGAMES*, 2004.

[14] Y. Kawahara, T. Aoyama, and H. Morikawa. A peer-to-peer message exchange scheme for large-scale networked virtual environments. *Telecommun. Systems*, 25(3):353–370, 2004.

[15] J. Keller and G. Simon. Solipsis: A massively multi-participant virtual world. In *Int. Conf. on Parallel and Distributed Process. Techn. and Applications, (PDPTA)*, 2003.

[16] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.

[17] J. C.-H. Lin and S. Paul. Rmtp: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, 1996.

[18] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, pages 84–95, 2002.

[19] G. Morgan, F. Lu, and K. Storey. Interest management middleware for networked games. In *Int. Symposium on Interactive 3D Graphics (SI3D)*, pages 57–64, 2005.

[20] PeerSim. `peersim.sourceforge.net`.

[21] M. Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra*, 2000.

[22] Quazal. `www.quazal.com`.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Int. Middleware Conf.*, pages 329–350, Nov. 2001.

[24] A. P. Yu and S. T. Vuong. Mopar: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV*, 2005.