DynFilter: Limiting Bandwidth of Online Games using Adaptive Pub/Sub Message Filtering

Julien Gascon-Samson, Jörg Kienzle, Bettina Kemme
School of Computer Science, McGill University
Montreal, QC H3A 0E9, Canada
Email: {Julien.Gascon-Samson}@cs.mcgill.ca,{Joerg.Kienzle,Bettina.Kemme}@mcgill.ca

***Abstract*—Multiplayer online games can generate a lot of server-related outgoing bandwidth, due to many factors such as highly variable amounts of players or the gathering of many players towards the same in-game locations. Predicting the exact amount of required bandwidth to support varying conditions can be costly, and players can experience game-wide failures if bandwidth is insufficiently provisioned. We present DynFilter, a game-oriented message processing middleware designed to adaptively filter state update messages for in-game entities located apart, in order to reduce bandwidth needs and stay within predefined quotas. We ran experiments on Amazon EC2 over a prototype game mimicking a FPS and a MMOG. Our results show that DynFilter is properly able to maintain bandwidth use within the pre-established quotas while still maintaining adequate delivery of relevant state update messages.**

I. Introduction

Multiplayer online games typically feature a large amount of players in the same shared virtual environment. Managing a large volume of players can be very challenging from a game operator's perspective, due to the fact that not only the game server, but also players must be aware of actions performed by other players or entities. Each action performed by any player or entity typically generates a game-specific state update message. Thus, as the number of players increases, bandwidth consumption within a game will grow at a faster-than-linear rate.

In order to maintain game immersion and gameplay quality, game state update messages must be delivered within specific time bounds and/or at specific frequencies. In First Person Shooter (FPS) games, messages are often delivered at a rate of about 20 updates per second due to the fast-paced nature of such games [14], [3]. FPS games feature smaller amounts of players. In Massive Multiplayer Online Games (MMOGs), update frequencies are much lower - a few updates per second; however, they usually have much larger amounts of players.

The amount of players in a given game at any given time is not constant: players are more likely to be playing at certain periods of the day or of the week [13]. Also, because MMOGs feature large-scale virtual worlds, players in such games often exhibit flocking behavior [11], [6], where a large amount of players gather towards the same popular locations on the map (towns, popular quests, etc.). Flocking can draw a lot of outgoing bandwidth from the servers since the number of messages that need to be transmitted within the flocking area increases in a near-quadratic way. If not handled properly, it can cause a game to collapse[1]. Thus, because of those phenomena, bandwidth use within a game is subject to variation over time.

Tile-Based Interest Management: If the size of the virtual world is large enough, it might not make sense for all players to receive state updates from all other entities in the whole virtual world. Interest management (IM) techniques are often used in games to limit the amount of messages that need to be transmitted by allowing players to receive updates only from players and entities located within a given area around them [4]. A typical approach is to split the game world into a set of tiles - tile-based IM - usually triangles, squares or hexagons. A player then receives updates for all other players located in her tile as well as in surrounding tiles, perhaps based on how far the player can see. Tile-based IM techniques alone, however may not be enough to support a sudden surge of additional players and become way less efficient if flocking happens.

Managing Bandwidth Needs: In order to support the variable outgoing bandwidth needs of online games, game operators typically provide their own server infrastructure. For games with a very large player base, a large amount of servers can be required. For instance, World of Warcraft, a popular MMOG, features over 250 servers[2] in order to support a playing base of millions of players. Bandwidth provisionning can be very challenging, since under-provisioning can cause the game to quickly become unplayable if a sudden burst of load is experienced[3]. Provisioning for the worst-case scenario will lead to servers being under-used most of the time.

With the advent of the Cloud, game providers may also opt to rent virtualized servers to benefit from increased scalability. However, outgoing bandwidth in a Cloud setting can incur significant expenses [2], especially when flocking happens. Game operators might wish to establish a maximum fixed budget that they are willing to spend per time period in order to better control and plan the costs of running their game, especially in the wake of indie or crowd-funded games.

DynFilter: In this paper, we propose DynFilter as a solution to above mentioned problems. DynFilter is a pub/sub-based middleware for online games which aims at limiting bandwidth

[1] http://www.gamespot.com/articles/blizzard-addresses-warlords-of-draenor-server-prob/1100-6423584/ [Aug 18, 2015]

[2] Decoupled into different game instances. Reference: http://us.battle.net/wow/en/status [Aug 18, 2015]

[3] http://www.cinemablend.com/games/World-Warcraft-Warlords-Draenor-Release-Marred-By-Server-Downtime-Lag-68412.html [Aug 19, 2015]

use by *filtering* update messages for entities located far away from each other in the game map, in order to respect target bandwidth constraints. The rationale behind DynFilter is based on the fact that it can be acceptable to discard some of the state update messages for entities that are located farther apart in the player's vision range without compromising the game play. Transparently discarding a portion of the state update messages can lead to significant reductions in bandwidth needs, and can lead to the following goals: (1) respecting a predefined Cloud-based bandwidth-related budget and (2) preventing the game from becoming unplayable due to a bandwidth use that would be above the allocated resources. In particular, DynFilter is based on channel-based publish/subscribe where tiles are considered as channels and players (and other relevant entities) are considered as subscribers / publishers, and where some of the state update publications can be dropped to lower bandwidth use.

This paper notably provides the following contributions:

- Game operators can define a maximum target outgoing bandwidth that they are willing to allocate over a given window as well as a maximum filtering (degradation of quality) that is allowed for each tile.
- Filtering is only applied to subscribers on remote tiles: state updates from players/entities located in the same tile (or group of tiles) are always delivered.
- A *load analyzer* module continuously monitors the pub/sub server with minimal overhead and analyzes the bandwidth that has been used in the current window. If needed, the *load optimizer* applies adaptive message filtering to reduce the amount of messages that need to be disseminated, in order to stay below the target bandwidth.
- Our algorithmic model automatically adapts filtering for each game tile based on the number of subscribers in the tile. Filtering is continuously recomputed.
- DynFilter is built as a thin layer over an unmodified pub/sub middleware (Redis [1]). It is also completely transparent and non-obtrusive to game players.

II. DynFilter Architecture

A. Tile-based Area-of-Interest and Message Delivery

In DynFilter, the game world is divided into a set of interconnected square tiles[4]. Assuming a world grid made of X columns and Y rows, we have a total of XY tiles labelled as follows: $T_{x,y}$ where $x \in \{0,\ldots,X-1\}$ and $y \in \{0,\ldots,Y-1\}$. Considering that a given player P is located in one and only one tile T_{x_p,y_p} at any given time, we define the subscription range Z as how many tiles *around* the player's current tile P will subscribe to in order to receive updates from other players and in-game entities. Formally, P will receive updates in all tiles $T_{x,y}|x \in \{x_p-Z,\ldots,x_p+Z\}, y \in \{y_p-Z,\ldots,y_p+Z\}$.

DynFilter makes sure that P receives all state updates in it's own tile (T_{x_p,y_p}). For surrounding tiles, state updates can be filtered, if needed. The impact of such filtering is greatly mitigated by the fact that players and entities located within these tiles are located farther apart from the player. Games typically make use of dead reckonning techniques [15], [10] to interpolate player positions between state updates. The inaccuracies in on-screen player positions (difference between dead-reckonned position and real position) will appear smaller for entities located farther away. Figure 1 gives an example of which tiles a given player located in the dark tile will be subscribing to (in that case, $Z = 2$): the black tile represents a subscription to it's own tile (unfiltered) and the grey tiles represent a subscription to surrounding tiles (can be filtered). Players are denoted as small dots.

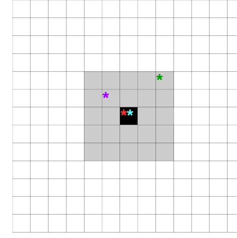


Figure 1: DynFilter Tiles Example

All messages are delivered using a publish/subscriber server that supports topic-based pub/sub, which may be run on the same machine as the game server, perhaps as part of the same process, for small-to-medium scale games. Alternatively, the pub/sub server can be run on a different machine, but in the same LAN or Cloud. Also, for large to epic-scale games, multiple pub/sub servers located on several VMs could be used to provide additional bandwidth [8].

B. Architectural Components

The DynFilter architecture is made of several distributed components. A high-level overview is presented in Figure 2, in a Cloud setting. A virtual machine (VM_Server) contains an instance of the pub/sub server (in our case, Redis, but any other suitable topic-based pub/sub server would be fine), coupled with a data collector component whose goal is to collect real-time data about all channels that currently exist on the pub/sub server, in a non-obtrusive way.

Aggregated data is periodically transmitted to the *load analyzer* module, which is located on a different VM, but in the same Cloud to reduce bandwidth overhead and costs (VM_LoadOptimizing). The *load analyzer* module determines if the allocated bandwidth quota for the current time period will be respected, based on previous bandwidth use and based on aggregate data received by the *data collector* (please refer to section III-A). A *load optimizer* module then computes a new *filtering matrix* (described at section III-B), which is transmitted to the *message filter* component, located on the same VM as the pub/sub server. The filtering matrix, which will be discussed later, is used to inhibit the delivery of some of the publications.

C. Message Filtering

In order to implement the described filtering behavior, DynFilter generates two pub/sub channels on the pub/sub

[4]Square tiles have been chosen because they simplify our spatial model. However, our model can easily be adapted to other tile configurations. For instance, if using triangular tiles, one could simply index each tile and substitute $F_{x,y}$ by a 1-dimensional array.

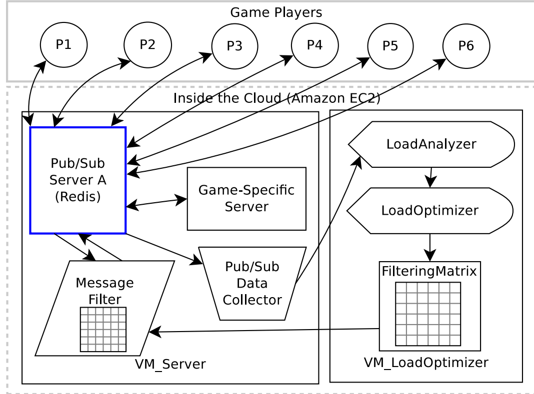


Figure 2: DynFilter Architecture Overview

server for each tile $T_{x,y}$, as follows: (1) $T^H_{x,y}$ (high-frequency - no filtering) and $T^L_{x,y}$ (low-frequency - filtering can occur). A given player P in tile T_{x_p,y_p} would subscribe to channel $T^H_{x_p,y_p}$ (receive all state updates in it's own tile) and would subscribe to channels $T^L_{x,y} | x \in \{x_p - Z, \ldots, x_p + Z\} \setminus x_p, y \in \{y_p - Z, \ldots, y_p + Z\} \setminus y_p$ (receive potentially filtered updates for all surrounding tiles). P always publishes to $T^H_{x_p,y_p}$. Therefore, no game entity or player will directly publish to any low-frequency $T^L_{x,y}$ channel.

DynFilter's message filter component is in charge of forwarding some or all of the publications from high-frequency channels ($T^H_{x,y}$) to low-frequency channels ($T^H_{x,y}$). It accomplishes that goal using the latest available filtering matrix $F_{x,y}$ that has been computed by the cost optimizer. In our model, $F_{x,y}$ is a 2-dimensional array that contains a filtering ratio for each tile $T_{x,y}$, between 0.0 (all messages are forwarded to $T^L_{x,y}$ - no filtering) and 1.0 (no message is delivered - this is not desirable so in practise, we provide an upper bound).

The message filter component subscribes to all high-frequency tile channels ($T^H_{x,y}$), which does not incur additional network overhead since it is local (on the same machine as the pub/sub server). For each high-frequency update at tile T_{x_0,y_0}, it does the following: (1) obtain the filtering ratio F_{x_0,y_0} from the filtering matrix $F_{x,y}$; (2) generate a random floating-point number between 0 and 1; (3) if the generated number is greater than F_{x_o,y_o} then (4) forward the publication to $T^L_{x_0,y_0}$ (if the generated number is smaller than F_{x_o,y_0}, then the message is not forwarded; thus, subscribers of $T^L_{x_0,y_0}$ will not receive it).

D. N-Layered Message Filtering

The DynFilter architecture is two-layered: the first layer (high-frequency) ensures full delivery of all messages and the second layer might allow for partial delivery. While it is possible to alter the various parameters such as the subscription range Z and the size of the tiles, for some games requiring finer granularity, DynFilter could be extended to introduce additional layers of message filtering to allow for a partial degradation in the amount of state updates received as the distance grows (future work). When playing at very high resolutions on large displays, players might want to be able to see objects located very far away. N-Layered filtering could be used to allow players to view remote objects at a very low update frequency.

III. Cost Analysis and Optimization

The main goal of the DynFilter load optimization process is to make sure that the allocated bandwidth quota for the current time period will be respected by filtering game update messages sent to subscribers of low-frequency tile channels.

A. Load Model & Analyzing

DynFilter defines the concepts of a *time unit* and a *time period*. A bandwidth quota (B_{quota}) is allocated for a given time period (10 minutes in our experiments) and is made of $t_{\max}$ time units (20 seconds in our experiments). B_{quota} is defined by the game operator (possibly by taking into account the outgoing Cloud bandwidth costs or the capabilities of it's current infrastructure). At every time unit t, the bandwidth that has been consumed since the beginning of the current period (B_{used}) is evaluated by the load analyzer. The load analyzer then computes the bandwidth that is *remaining* until the end of the period: $B_{\text{remaining}} = B_{\text{quota}} - \text{B}_{\text{used}}$.

From the remaining bandwidth, a *target* bandwidth allocation is then computed for the next unit, which is the average amount of bandwidth that the game should consume in all remaining time units throughout the end of the period, in order to respect B_{quota}. It is defined at equation 1.

$$B_{\text{target}} = B_{\text{remaining}} / (t_{\max} - t) \tag{1}$$

The next step is to determine if, by consuming bandwidth at the *current* rate, the game would go over B_{quota}. To do so, the load analyzer first considers the bandwidth that has been consumed in the last unit (B_{prev})[5]. If $B_{\text{prev}} \leq B_{\text{target}}$, then filtering can be reduced or cancelled if it is no longer needed, since we are *currently* using less bandwidth than allowed. However, if $B_{\text{prev}} > B_{\text{target}}$, then we are *currently* using too much bandwidth, and we need to lower bandwidth use. We define B_{remove} as the bandwidth that we need to remove in the next time unit as follows: $B_{\text{remove}} = B_{\text{prev}} - B_{\text{target}}$. By knowing how much bandwidth we need to remove, the load optimizer then computes an appropriate filtering matrix, as explained at the following section.

B. Load Optimization

The filtering ratio $F_{x,y}$ for tile $T_{x,y}$ was defined as the ratio of messages that should not be delivered to $T^L_{x,y}$.

Trivial filtering: Assuming for simplicity that there was no filtering in the previous time unit, by knowing the number of bytes to remove (B_{remove}) as well as the number of bytes consumed in the previous time unit (B_{prev}), we can compute one global filtering ratio F for all tiles as follows: $F = \frac{B_{\text{remove}}}{B_{\text{prev}}}$. If there was already filtering in place, then we need to compute an *extrapolation* of the bandwidth that would have been consumed in all low-frequency tiles over the last unit if no filtering was in place, by inverting the effects of the filtering already in place, following a similar process as

[5] We initially considered using the averaged bandwidth over all time units since the beginning of the period. We found out that this approach worked well only if the bandwidth did not vary too much. By taking the bandwidth over the last time unit only, we are able to react quickly to sudden variations.

described in equation 2. We would then obtain an extrapolated version of B_{remove} and B_{prev} which would yield an accurate computation of F.

We want to go beyond trivial filtering and consider the specificites of each tile as per the following rationale: filtering can be stronger on tiles with many players since the updates of any individual player will be less apparent in a crowd. In addition, players usually closely follow only a limited amount of players at the same time and pay less attention to the others [3], [14]. On the contrary, filtering should be lower on tiles with fewer players. DynFilter proposes an algorithm to compute a varying filtering ratio for each tile while still respecting bandwidth quotas.

DynFilter filtering: In order to obtain a filtering ratio for each tile $T_{x,y}$, we need to determine how many bytes we should save for every low-frequency $T^L_{x,y}$ tile channel. The idea is that the number of bytes that we should *remove* from each tile channel should be proportional to the total outgoing bytes of that tile channel for the previous unit. However, we multiply the number of bytes to be removed by a density factor $D_{x,y}$ that is logarithmic to the number of subscribers in that tile, to take the number of subscribers in the tile into account.

Let $S_{x,y}$ be the number of subscribers in $T_{x,y}$, $B^H_{x,y}$ the outgoing bandwidth (over the previous time unit) of channel $T^H_{x,y}$ and $B^L_{x,y}$ the outgoing bandwidth of $T^L_{x,y}$. Again, as it was the case with trivial filtering, $B^L_{x,y}$ depends on the filtering ratio $F_{x,y}$ used in the last time unit. In order to get accurate bandwidth computations, we define $B^{*L}_{x,y}$ as the *extrapolated* outgoing bandwidth, which is a projected value of $T^L_{x,y}$ without the effects of filtering (equation 2).

$$B^{*L}_{x,y} = \frac{B^L_{x,y}}{1 - F_{x,y}} \tag{2}$$

For every tile, we compute the density factor $D_{x,y}$ as follows: $D_{x,y} = \log_2 S_{x,y}$. We then compute the weight factor for tile $T_{x,y}$ by multiplying the total *extrapolated* bandwidth with the density factor (equation 3).

$$W_{x,y} = (B^H_{x,y} + B^{*L}_{x,y}) \cdot D_{x,y} \tag{3}$$

We define the sum of the weight factors as follows: $W_T = \sum W_{x,y}$. For each tile $T_{x,y}$, we can then compute how many bytes we need to remove from channel $T^L_{x,y}$ (equation 4).

$$Q_{x,y} = (W_{x,y}/W_T) \cdot B_{\text{remove}} \tag{4}$$

By knowing $Q_{x,y}$, we compute the filtering ratio for tile $T_{x,y}$ using equation 5 (ratio of bytes to remove to the number of *extrapolated* outgoing bytes that flowed through $T^L_{x,y}$ over the last time unit).

$$F_{x,y} = \frac{Q_{x,y}}{B^{*L}_{x,y}} \tag{5}$$

Note that a maximum value can be set for $F_{x,y}$ so that we can ensure that a minimal ratio of state update messages will be forwarded (in our experiments, it is set to 0.75) so at least 25% of the state update messages will be sent accross all tiles.

The load optimizer then transmits the matrix of all filtering ratios to the message filter component that applies it.

IV. Experiments

A. Implementation and Experimental Setup

We implemented DynFilter in Java on top of the Dynamoth pub/sub-based message dissemination infrastructure. Pub/sub is provided by unmodified Redis middleware (open-source). We ran our experiments on DynGame, which is a prototype game skeletton that reuses some components of the Mammoth project [9]. DynGame can support a large amount of players that randomly move and uses square tiles. Publications and subscriptions are made according to the DynFilter model.

Experiments have been run in the Cloud over a set of 20 Amazon EC2 instances: one *m3.medium* instance for the pub/sub server, data collector and message filter components; one *m3.medium* instance for the load analyzer and the load optimizer; one *t2.micro* instance for experimental data collection and 17 *t2.micro* instances to run our game players. We determined that we were safely able to run 15 players per instance, up to a maximum of ~250 players. While our implementation has been designed to support multiple pub/sub servers, we decided to limit our experiments to only one pub/sub server for simplicity reasons.

We considered a subscription range $Z = 2$ (players subscribe to 25 surrounding tiles, please refer to Figure 1), except for players located near edges. The subscription to the central tile is at high-frequency ($T^H_{x,y}$) (all messages are received) and the subscriptions to the other tiles is at low-frequency ($T^L_{x,y}$) (messages can be dropped).

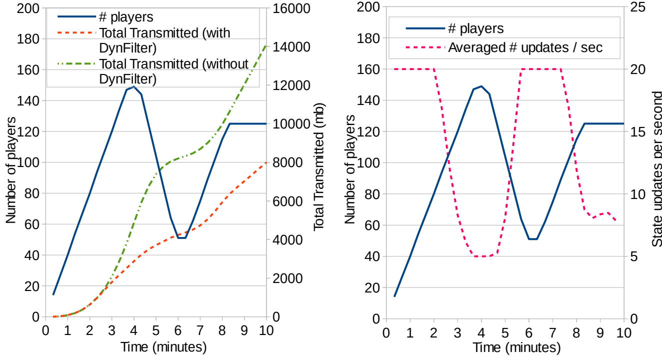
B. Experiment 1: FPS Game / Scalability

Description: The goal in this experiment was to assess the scalability of DynFilter and its bandwidth-limiting capabilities in the context of a FPS-like game with many players. A typical FPS has a limited amount of players because of the high-bandwidth that is needed to support the high frequency of updates and the vision range for all players. For instance, WatchMen, which was based on a modified version of Quake 3, supported up to 48 players in the same game [14] (the original Quake 3 supported only 16 players).

We setup a map with up to 150 players and 100 tiles (10x10), with $Z = 2$. That means that any player would be able to view up to 25% of the map at any given time, which makes sense since FPS maps are generally smaller-scale. As mentioned in the introduction, because of the fast-paced nature of FPS games, players optimally receive up to 20 state updates per second.

We progressively injected up to 150 players in the game, then reduced to 50 players, then increased again up to 125. We allocated a bandwidth threshold of 8000 Mb for the duration of the period (10 minutes), with units of 20 seconds (load analyzing and optimizing occured every 20 seconds).

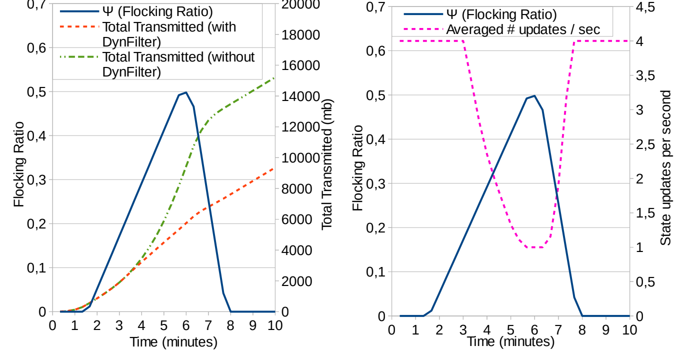
Results: Figure 3 details our results for the FPS experiment. On Figures 3a and 3b, until about 3 minutes, we can see that no filtering occured on low-frequency tile channels (all state updates are transmitted). Afterwards, due to the highly increasing load caused by the large amount of players, filtering starts to occur. The frequency of state updates received on low-frequency tile channels progressively drops



(a) Number of Players and Total Outgoing Bandwidth

(b) State Updates per Second

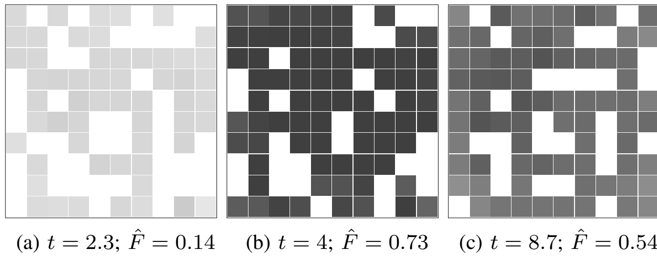
Figure 3: FPS Game / Scalability Experiment Results



(a) Flocking Ratio (ψ) and Total Outgoing Bandwidth

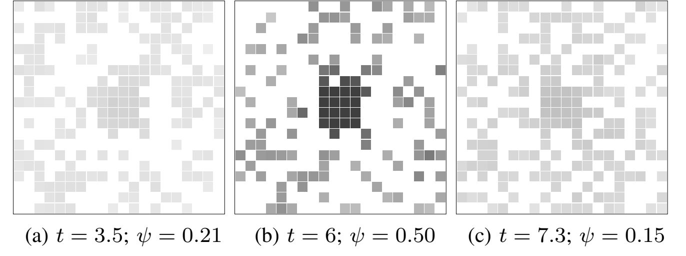
(b) State Updates per Second

Figure 5: MMOG Game Experiment Results



(a) $t = 2.3$; $\hat{F} = 0.14$ (b) $t = 4$; $\hat{F} = 0.73$ (c) $t = 8.7$; $\hat{F} = 0.54$

Figure 4: Filtering Ratio Heat Map / FPS Game



(a) $t = 3.5$; $\psi = 0.21$ (b) $t = 6$; $\psi = 0.50$ (c) $t = 7.3$; $\psi = 0.15$

Figure 6: Filtering Ratio Heat Map / MMOG Game

until it reaches an average of 5 updates per second, which is the minimum frequency allowed for this experiment. Despite having a reduced frequency, player avatars will still appear to be fluid since updates will still be received at least at every 200 ms, and filtering only applies to players being farther appart (in different tiles). As mentioned previously, dead reckonning will compensate for some missing updates.

We then observe that as the number of players starts to shrink (at about 4 minutes), the number of updates per second start to raise again until it reaches 20, which means that no filtering occurs - all state update messages are delivered to low-frequency tile channels. Then, as the number of players raise again above a certain threshold and up to 125, the number of updates per second starts to shrink again down to ~8 updates per second, which is a *best compromise* on degradation that will ultimately lead to a total bandwidth use of 8000 Mb at the end of our period; thus, respecting our predefined bandwidth quota. Overall, 8000 Mb have been used instead of 14000 Mb, thus representing a bandwidth saving of 43%.

Figure 4 illustrates the *averaged* filtering ratio ($\hat{F}$, which is the filtering ratio that would be equivalent to the current global reduction in bandwidth if all tiles had the same filtering ratio) for all 100 tiles, at time snapshots $t = 2.3$, $t = 4$ and $t = 8.7$. White means that no filtering is in effect for a given tile (or no player is in that tile), dark grey means that filtering is at up to 75% and intermediate shades of grey illustrate an intermediate filtering ratio. We notice that as the number of players increases, the filtering ratios increase in roughly the same way accross the whole game map (except for tiles with no players) since the density of players is similar.

C. *Experiment 2: MMO Game with Flocking*

Description: In this experiment, we wanted to assess how DynFilter was able to handle the case of flocking within a medium-scale MMO game. Flocking refers to situations were many players gather towards the same location on the map, which can put a strain on the system since the number of state updates to be transmitted grows quadratically.

Since MMOGs are slower-paced games compared to FPS, we settled for an update rate of 4 updates per second. Also, since game worlds are much larger, we opted for 400 tiles (20x20). Thus, with a subscription span of 25 tiles, players only see a maximum of 6.2% of the map. We allocated a bandwidth quota of 10000 Mb for the 10-minutes period. We quickly injected 250 players in the map (which in itself would not go above the quota; thus, would not trigger the use of filtering). In this experiment, when flocking, a given player will be quickly moving towards the center 4x4 tiles of the map and will remain within those tiles.

After injecting 250 players, we slowly increased the *flocking ratio* (ψ) from 0 to 0.5, which meant that up to 50% of the players were eventually located in the 16 centric tiles, thus greatly increasing player density and the number of messages that the pub/sub server had to deliver (near-quadratic growth).

Results: Figure 6 describe our results for the MMOG experiment. At time $t = 1$, ψ slowly starts to increase. After 3 minutes, when ψ reaches ~30%, DynFilter starts applying filtering in order to reduce the amount of messages that need to be transmitted and thus, the bandwidth use. In Figure 5b, we observe that the average number of state updates per second for low-frequency tile channels starts to reduce until it reaches 1

(minimum allowed in this experiment), in order to compensate for the drastic increase of bandwidth. At $t = 6$, ψ slowly starts to decrease (players stop flocking and slowly move elsewhere to a random location anywhere in the map). In reaction to the reduction in bandwidth use, the average number of state updates per second starts growing again until it reaches 4 (low-frequency tile channel filtering disabled).

At the end of the quota, a bit less than 10000 Mb had been used since in the last minutes, we transmitted messages at the same rate as high-frequency tile channels, which led to using less than the allowed quota. Overall, DynFilter was able to save 38% of the bandwidth.

Figure 6 shows a snapshot of the distribution of the filtering ratios accross all tiles. At time $t = 3.5$, when flocking slowly starts to happen; the load optimizer starts increasing filtering ratios globally with a small emphasis on the center tiles. At time $t = 6$, filtering gets more important and really more concentrated in the center of the map. At time $t = 7.3$, when flocking is being reduced, we observe that flocking ratios in the center tiles gets less emphasized. This Figure showed that DynFilter was able to adjust it's filtering based on the density of the tiles, in order to ensure that tiles with a lower amount of players would keep sending updates at a higher frequency despite the overall reduction in bandwidth, to account for the fact that players are more likely to notice individual players in lower-density tiles compared to higher-density tiles.

V. Related Work

To the best of our knowledge, no work has been done in the field of automatically filtering/throttling publications in topic-based pub/sub systems; therefore, we think that our work is a novel approach and that it is well suited for games. Also, very few pub/sub systems optimized towards games have been proposed. The Mammoth massively multiplayer game framework [9] features a network engine that proposes a pub/sub-like topic-based interface. Dynamoth [8] is a scalable topic-based pub/sub middleware oriented towards applications with tight latency requirements such as games. In [5], the authors evaluate how different pub/sub architectures (channel-based and content-based) can be used in the context of MMOGs to efficiently process message delivery as well as some other game-related tasks such as interest management.

Restricting Bandwidth : WatchMen [14] and Donnybrook [3] propose mechanisms to reduce bandwidth use in P2P-based FPS games by reducing the rate at which updates are delivered for players located outside of other player's vision range. More specifically, full updates are sent only to the *k*-most interested players within a given player's vision range. Players located in the vicinity but not in list of *k*-most interested players receive different, less frequent messages, that contain data used to aid in performing dead reckonning. Finally, players located very far receive sporadic state update messages. Thus, these approaches do not only vary the frequency of message delivery, but also the contents of state updates. In contrast, we adjust the update frequency depending on the number of players in the vicinity. Some other P2P schemes have also been proposed to reduce bandwidth while supporting varying conditions [16]. A drawback is that P2P is prone to less-predictable latencies and bandwidth capabilities.

Some load-balancing MMO game architectures have been developed to try to mitigate the impacts of flocking in terms of bandwidth needs by dynamically reassigning load to different servers when needed [6]. However, they do not lead to a decrease in bandwidth needs; thus, costs can still be significant.

Cloud Gaming: Cloud gaming has been a hot research topic in the last few years. It involves running whole games (clients and servers) in the Cloud: game players play using thin clients that stream a live video of the game and transmit user input back to the server. While there are interesting properties such as increased security and the ability to play on multiple devices with minimal porting efforts, there are also drawbacks such as high Cloud bandwidth use (might be expensive) and high client bandwidh use, which might be limited and might be very costly on mobile data plans, despite using degradation techniques [7]. Latencies will also be higher [12], which might be problematic for latency-sensitive games such as FPS.

VI. Conclusion

We proposed DynFilter, a middleware designed to adaptively filter game state update messages in order to limit bandwidth use within a game to a predefined threshold. A major contribution is that our platform does per-tile filtering in order to adjust filtering levels to the number of players in each tile. We ran experiments in the context of FPS games with a high-frequency of updates and in the context of MMOGs with flocking. In both cases, DynFilter was correctly able to limit bandwidth use while maintaining the normal flow of the gameplay. As future work, we would like to extend our model to include N-Layered filtering.

References

[1] Redis website (2013), `http://www.redis.io/`
[2] Amazon EC2. `http://aws.amazon.com/en/ec2/pricing/` (2015)
[3] Bharambe, A., Douceur, J.R., Lorch, J.R., Moscibroda, T., Pang, J., Seshan, S., Zhuang, X.: Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In: ACM SIGCOMM 2008 Conf. on Data Communication. pp. 389–400
[4] Boulanger, J.S., Kienzle, J., Verbrugge, C.: Comparing interest management algorithms for massively multiplayer games. In: NetGames 2006
[5] Cañas, C., Zhang, K., Kemme, B., Kienzle, J., Jacobsen, H.A.: Publish/subscribe network designs for multiplayer games. In: Middleware 2014. pp. 241–252 (2014)
[6] Chen, J., Wu, B., Delap, M., Knutsson, B., Lu, H., Amza, C.: Locality aware dynamic load management for massively multiplayer games. In: PPoPP 2005. pp. 289–300 (2005)
[7] Claypool, M., Finkel, D., Grant, A., Solano, M.: Thin to win? network performance analysis of the onlive thin client game system. In: NetGames 2012. pp. 1–6 (2012)
[8] Gascon-Samson, J., Garcia, F.P., Kemme, B., Kienzle, J.: Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In: ICDCS 2015. pp. 486–496 (June 2015)
[9] Kienzle, J., Verbrugge, C., Kemme, B., Denault, A., Hawker, M.: Mammoth: a massively multiplayer game research framework. In: Foundations of Digital Games (FDG). pp. 308–315 (2009)
[10] Pantel, L., Wolf, L.C.: On the suitability of dead reckoning schemes for games. In: NetGames 2002. pp. 79–84 (2002)
[11] Pittman, D., GauthierDickey, C.: A measurement study of virtual populations in massively multiplayer online games. In: NetGames 2007. pp. 25–30 (2007)
[12] Shea, R., Liu, J., Ngai, E.H., Cui, Y.: Cloud gaming: architecture and performance. Network, IEEE 27(4), 16–21 (July 2013)
[13] Tarng, P.Y., Chen, K.T., Huang, P.: An analysis of wow players' game hours. In: NetGames 2008. pp. 47–52 (2008)
[14] Yahyavi, A., Huguenin, K., Gascon-Samson, J., Kienzle, J., Kemme, B.: Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In: ICDCS 2013. pp. 134–144 (July 2013)
[15] Yahyavi, A., Huguenin, K., Kemme, B.: Interest modeling in games: The case of dead reckoning. Multimedia Syst. 19(3), 255–270 (Jun 2013)
[16] Yahyavi, A., Kemme, B.: Peer-to-peer architectures for massively multiplayer online games: A survey. ACM Comput. Surv. 46(1), p1–51 (2013)