GSM: A Game Scalability Model for Multiplayer Real-time Games

Jens Müller and Sergei Gorlatch
University of Münster, Germany
Email: {jmueller|gorlatch}@uni-muenster.de

Abstract— **Current commercial real-time computer games increasingly provide game designs suitable for a high number of players, forming the class of Massive Multiplayer Games (MMG). To support MMG, the *scalability* of network topologies becomes critically important, i.e., their ability to maintain the game service for an increasing number of players. This paper presents GSM - an analytical scalability model for a detailed investigation of massively multiplayer capabilities of different networking topologies. We use the GSM to discuss the suitability of the client-server and peer-to-peer topology, as well as our own concept of a multi-server topology, for the important classes of First Person Shooter (FPS) and Real-time Strategy (RTS) games whose designs are still rarely adopted in MMGs. We verify our analytical model in scalability experiments in which we were able to forecast the maximum numbers of players in a non-congested game session with an error of less than 7 %.**

I. Introduction

In a real-time computer game, the overall quality of the game service depends on different factors like fault tolerance, game responsiveness and prevention of player cheating. These aspects are heavily influenced by the networking architecture used in the game application. When it comes to massively multiplayer games, the provided level of scalability of a particular network topology is an especially important factor. In this paper, we present a novel analytical model for scalability which allows to compare the suitability of different network topologies for massive multiplayer game designs.

Remarkable research effort in the area of scalable network topologies has been made for the genre of *Massive Multiplayer Online Roleplaying Games (MMORPG)*. In such a game, a huge game world is provided for thousands of participating players by a multi-server topology, in which partitions of the game world usually are distributed amongst servers for maintenance [1]. Unfortunately, this topology is not applicable to the genres of *First Person Shooter Games (FPS)* and *Real-time Strategy Games (RTS)*, as discussed in Section II-F.

For commercial FPS and RTS games, networking architectures have not been changed for years. Commonly used network topologies are client-server and peer-to-peer which are comparatively easy to implement, but suffer from poor scalability. This becomes increasingly critical for team-based FPS games like *Counterstrike* [2], which currently evolve towards large-scaled military battle simulations taking place in a huge area with vehicles and aircrafts like in *Operation Flashpoint* [3] or *Battlefield 1942* [4]. Such game designs support a high number of users, but the bottleneck in form of the single server of the commonly used client-server topology limits current FPS games to 64 participating players at maximum. The peer-to-peer topology, as used in the RTS game *Age of Empires* [5] for example, supports even less participating players in a game session. In general, the potential of these classes of games cannot be fully exploited without a scalable network architecture.

An alternative to the client-server and peer-to-peer concepts is the approach of a proxy server-network for FPS and RTS games [6] which aims at improved scalability and enhanced responsiveness of massive-multiplayer sessions using multiple servers rather than a single one. While first experiments with this new architecture showed promising results in terms of a higher number of participating users, a detailed comparison to known concepts is necessary.

In this paper, we present the GSM - *Game Scalability Model*, an analytical model for studying the scalability of network topologies in multiplayer real-time games. Our main goals in developing this model are as follows:

- **Analytical topology comparison** - We want to provide a possibility to compare the scalability of different networking topologies for massive multiplayer games. This comparison should be general enough, i.e., independent of a certain game implementation, in order to be valid for whole classes of games rather than for a particular game.
- **Decision guidance** - The results of the comparison based on the model should provide game developers with a detailed decision guidance to determine which topology performs best in terms of scalability and responsiveness for different game designs. The model therefore should base on variables which can be estimated or obtained from experimental measurements in an early stage of the game developing process.
- **Self monitoring of resource usage and load balancing** - Although the model allows a comparison of topologies independently from a particular game application, it should also be possible to incorporate the model into a particular game. This way, a self monitoring of utilized resources and a prediction of maximum player numbers at runtime is possible. In network topologies which use several servers for a single game session, such a monitoring could be used for automatic load balancing during the game by migrating clients of highly loaded hosts to servers with low working load.

Previous work dealt with resource utilization in different network topologies for computer games. In [7], the bandwidth utilization of different network topologies for computer games has been discussed, focusing on the consistency of the state of the game. In order to compare the scalability of network topologies, the computational load at the participating hosts has to be taken into account as well, as we do in our model. A good overview of different networking topologies is provided by [8], but it does not provide the level of detail we need for the scalability analysis.

The remainder of this paper is organized as follows: In Section II, we define scalability in the computer games context and provide an analytical scalability model for each of the different topologies. We discuss the scalability characteristics of the three different topologies using the GSM analytical model in Section III. In Section IV, we present an experimental verification of the GSM. We compare our work to related activities and conclude the paper in Section V.

II. Modeling Game Scalability

In the distributed setting of an online computer game, two different types of components are involved: On the local computer of each participating user, a *client component* is responsible for rendering graphics, playing music and sound effects and reading in user input. There is also at least one *server component* which receives user inputs, computes changes to the game world and sends updates of the virtual world to participating client components. The architecture of the distributed system determines the number and location of server components, the distribution of tasks among the participating components, and the communication links between them. While the general concept of two types of components exist in different architectures for multiplayer computer games, the architecture-dependent communication setup and distribution of computational tasks differs a lot between different architectural approaches.

In the context of online computer games, *scalability* is the ability of a distributed game architecture to maintain service when the number of participating players increases. For the users, the overall game experience and immersion in the game, i.e. the feeling to be actually inside the virtual game world, requires a well performing system in various aspects: low lag and packet loss in game communication, fast rendering of graphics, etc. However, the scalability of the distributed gaming architecture itself only depends on the availability and usage of two resources: computing cycles and network bandwidth of the server components. In a client-server setup, only the single server component is of interest when considering scalability, while in a peer-to-peer session each participating host runs a client as well as server component, usually in the same operation system process. Therefore, the usage of these two resources has to be investigated for each participating peer. The utilization of computing cycles as well as network bandwidth commonly increases with a growing number of participating users. Therefore, up from a certain number of users, more resources are required for the game processing than available, which results in congestion of the server component on one or more host machines. In order to compare the usage of these resources in different network topologies, we first analyse how a real-time computer game is processed.

A. Processing a Real-time Game

A real-time game is a continuous application, but the game simulation, i.e. its processing by the participating components, is usually done at certain points in time which discretize the application.

During the processing of a real-time game, the *state* of the application, i.e. the state of all dynamic entities in the game like their position in the game world and the currently performed action, is periodically altered by the server components. Therefore, we model the processing of the game over some period of time as a sequence of state transitions $\phi_{S_i,S_{i+1}}$. At each transition, the new state S_{i+1} is calculated from the current state S_i and the user inputs occurred during the time the game simulation was in the state S_i. In this definition, a state S_i is valid for a certain period of time while ϕ describes the calculation for state transitions. The new state S_{i+1} has to be at least partially transferred to other components after the calculation of $\phi_{S_i,S_{i+1}}$.

In general, the amount of computing cycles required for the calculation of ϕ as well as the bandwidth utilized for transmission of updated state information increase with the number of participating client components, because the number of user inputs per time to process and the number of updates per time to be transferred to client components increases. The frequency at which state transitions are performed is usually referred to as the *tickrate* or *processing frame rate*. In order to provide a fluent game experience, there is a maximum length of time after which client components are expected to receive an updated game state from the server component. Therefore, computer games which provide users the illusion of continuous game interactions can be viewed as soft real-time systems. For the distributed system, this maximum length of time for the calculation of a new state can be given either explicitly or implicitly.

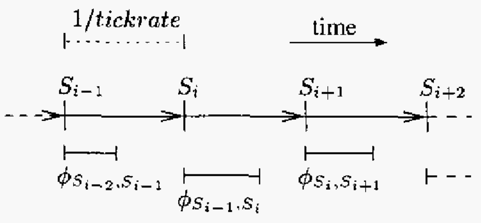
If the maximum length of time for a state update is given explicitly, the server component knows this value at runtime. This time can either be given by a user during session setup by specifying the tickrate for the session as in the client-server FPS game *Unreal Tournament* [9] or automatically computed by the server components as in the peer-to-peer RTS game *Age of Empires* [10]. This way, server components know the timing constraints of the real-time application and can measure possible occurrences of lateness in calculation. Additionally, since game servers usually do not run on real-time operating systems, the tickrate can be adjusted to the behaviour of different hardware platforms and operating systems to provide an overall equal level of game responsiveness and fluentness. *Unreal Tournament* servers, for example, usually run with a tickrate of 35 updates per second on Windows, while the tickrate on Linux (Kernel 2.4) should be set to about 50-55 updates per second to achieve the same game responsiveness.

With an implicitly given timing constraint for the calculation of a new state, server components provide a best effort service in which client requests are answered as fast as possible with replies containing the updated game state. In request-reply based computation scheme, the server does not know the maximum length of time in which request have to be calculated and therefore can not determine whether the calculations are late with respect to the real-time constraint. The maximum time for calculations is implicitly given by game design aspects in combination with commonly used prediction algorithms for latency hiding on client components [8], [11], which inter- or extrapolate positions of dynamic game objects on the client side to provide fluent movements. However, such algorithms expect to periodically receive the real position from the server to compensate prediction errors. If an update is delayed, the difference between the real and predicted position grows and eventually leads to warping of dynamic objects. For example, the client of the client-server FPS game *Quake* [12] expects to receive updates every 30 to 40 ms (corresponding to a tickrate of 25 to 33 updates per second), while the server itself does not know about any timing constraints. Therefore, although no explicit maximum length of time for the calculation of a new state is given at the server in the request-reply based computation scheme, the distributed system with all participating processes is a real-time system and implicitly inherits assumptions about the maximum time for state updates.

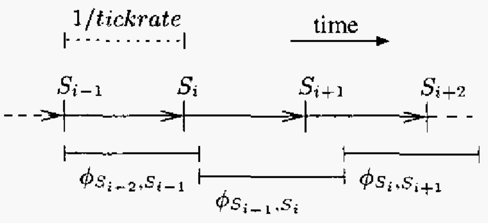
The general observation for the processing of a real-time game is that there always exists an maximum amount of time for the calculation of a new state, given either implicitly or explicitly. Therefore, we model the distributed game application as a soft real-time system with a maximum length of $1/tickrate$ seconds in time for a computation of a new state executed by server components which perform $tickrate$ of these state updates per second.

The total number of supported client components, i.e., the game clients processes, is limited by the available resources for running the server components. If a server component is not able to finish calculation of ϕ in the available time of $1/tickrate$ seconds, this component is computationally congested and the game simulation cannot proceed properly. Although there are possibilities to adjust a tick's length with respect to the computational performance as done in the peer-to-peer RTS game *Age of Empires* [10], we consider the length of a tick to be constant: This static length can be set to the maximum length possible without lowering the targeted responsiveness of a game, i.e. the maximum time for a user action to be processed and displayed. Having the ticks' length already set to the maximum value this way, a dynamic length extension of a tick would have the same effect as an unintended host congestion and therefore has to be avoided in order to maintain the targeted style of game play and responsiveness.

Figure 1 depicts two example transition sequences in the processing of a game. In 1(a), the processing is correct: the



(a) Correct game processing



(b) Congested game processing

Fig. 1. Example transition sequences

calculations of the transitions ϕ finish in the available time of $1/tickrate$. The calculation at the process depicted in Figure 1(b) is congested: The calculation of ϕ takes longer than the length of a tick, resulting in an incorrect game processing which violates the timing constraints of the real-time simulation.

In addition to being *computationally congested*, a process can also be *communicationally congested* if the available bandwidth at the host machine is not sufficient to send and receive updated state information in the available time. Both available computational and communicational resources determine the maximum number of participating clients, which also heavily depends on the underlying communicational topology and computational distribution of the calculation of ϕ.

B. Worst Case Consideration

It is obvious that the length of the calculation of ϕ depends on the current gaming situation. If most of the players are scattered in the game world and only few *avatars* (the virtual representations of participating users) are visible to each other, then few interactions have to processed in the calculation of ϕ. Otherwise, if nearly all participating players perform an action and are able to see many other avatars, much more calculations have to be done during a single tick at server components.

The latter situation commonly occurs at fights between a lot of players in FPS games, where all players are able to see each other and simultaneously perform movement or attack actions. In RTS games, the worst case commonly occurs at huge battles with a lot of units, where players are able to see nearly all units of the game and issue a lot of commands to their own units. It is very likely that the winners of such a huge battle will win the complete match or at least gain an advantage. The game has to be fully responsive especially when such large fights occur: If the game processing is congested in such situations, the outcome of the battle is decided not by users' skill but accidentally, which is not acceptable. In order to provide a correct game processing in all situations, such a *worst case* has to be considered for the scalability analysis. Unfortunately, there is no generic worst case for all game designs and game worlds. If there is, for example, only a single central place in the game world where to get good game items like special armor or weapons, the worst case consists in interactions between all players which will eventually meet there. If there are several such spots scattered throughout the map, the probability for a big clash of all players is much lower and the worst case does not involve interactions between all players.

In our model, we assume that the worst case consists in interactions of **all** participating users. This way, we ensure a correct game simulation in all cases regardless of the particular game design. However, the model can easily be adapted to specific game designs in which the worst case for computation time does not include interactions of all players, as we discuss in III-D.

C. The Game Scalability Model

In our model for scalability, we consider a game session with n participating clients and m game-controlled entities. Such entities are the dynamic parts of the game not controlled by participating users, with which, however, the avatars of the users can interact. Examples for server entities are items which can be picked up by avatars, and *Non Player Characters (NPC)* which are avatars controlled by the game itself.

The general state computation scheme of a single tick at a server component is as follows:

1. Receiving, validating and processing of all user actions issued for this tick. A user action is one of all the possible game commands like moving oder interacting with other avatars.
2. Processing of game-controlled entities like NPCs or item respawns.
3. Filtering, i.e. determining the visible state changes for a particular client, and sending of updated state information to clients.

Although this scheme seems to be simplistic, it is in fact the realistic computation scheme used in sophisticated computer games like *Quake* [13], *Half-Life* [14] or *Unreal Tournament* [9] for a single tick.

We assume that all network topologies analysed in the following use the same data model to represent the state of the game as well as the same algorithms to modify the state. Therefore, although the computational and communicational tasks are distributed across the participating processes in very different ways in the different topologies, a single basic subtask utilizes the same resources throughout all topologies discussed here. This way, different topologies can be fairly compared to each other.

D. Client-Server Network Topology

In the client-server topology depicted in Fig. 2, the server is the only process which modifies the game state. Therefore, all calculations required for a state transition ϕ have to be done at the single server component, which has to receive user inputs from each of the participating clients and to inform the clients about the updated state at each tick. In our model, we only consider the resource usage of the server component running on a dedicated host, because the participating clients do not limit the maximum number of participating players as long as the clients are able to receive the updated state informations in time. Even if the amount of data sent to a client exceeds its downstream bandwidth of the communication interface, a prioritization mechanism omitting information of less interest can reduce the required data volume. Such a mechanism is already implemented in many FPS games (e.g., in *Unreal Tournament* [9]) in order to allow the participation of hosts connected via low bandwidth interfaces such as modems.

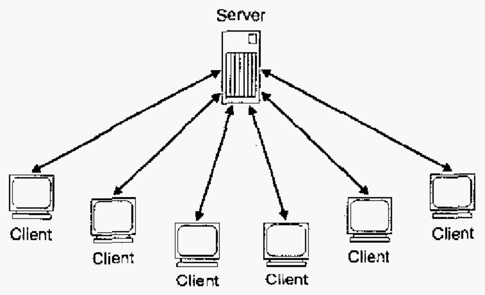


Fig. 2. Example of the client-server topology with six clients

The tasks for the server in the calculation of a single state transition and the computational time required for these tasks are as follows:

- Receiving, validating and processing of n client actions, which requires the time for processing $t_{ca}(n, m)$ (*ca* denotes *client action*) for each client in the worst case. The time $t_{ca}(n, m)$ itself depends on the actual values of n and m, since interactions of a certain player could make it necessary to process a set of other players' data or server entities. An example for such multiple processing of avatars is a player firing a rocket into a group of opponents. For each of the hit avatars, the perceived damage has to be calculated and applied. The amount of data received from a client in a single tick has a maximum static size denoted by d_{cin} (*client in*).
- General processing of the game world which does not depend on user actions at the specific state transition. Examples are periodic item spawns, resource harvesting or actions performed by NPCs. In our model, we assume for each tick that m game-controlled entities have to be processed with a maximum time of $t_{se}(n, m)$ (*server entities*) for each entity. While some processing takes constant time independent of the number of players n, e.g.,

timed item spawns, other computations like calculating the behaviour of NPCs depend on the number of players and entities.

- Filtering and transmission of the new state S_{i+1} to clients. In order to prevent cheating at the clients, only state information directly visible to players is transmitted [15]. Although this filtering requires additional computation at the server, it also reduces the amount of data sent to clients and, therefore, the amount of outgoing bandwidth required by the server. The filtering of data sent to a single client takes a time of $t_{fc}(n, m)$ (*filtering client*) in the worst case, while each message sent has a maximum size of $d_{cout}(n, m)$ (*client out*). Both the time required for filtering and the amount of data sent depend on the total number of players and server entities in the session.

Altogether, the calculation of a single tick in the client-server (*cs*) topology takes a time of

$$T_{cs}(n, m) = n \cdot t_{ca}(n, m) + m \cdot t_{se}(n, m) + n \cdot t_{fc}(n, m) \quad (1)$$

We summarize utilized bandwidth separately for incoming and outgoing traffic. This way, it is possible to provide an accurate analysis of what kinds of Internet connections (asymmetric DSL, full duplex backbone connection etc.) are suitable. The utilized bandwidth for incoming traffic at the server host results in

$$D_{cs}^{in}(n) = n \cdot d_{cin} \quad (2)$$

while the outgoing traffic utilizes the bandwidth of

$$D_{cs}^{out}(n) = n \cdot d_{cout}(n, m) \quad (3)$$

E. Peer-to-Peer Network Topology

In the classical peer-to-peer topology, depicted in Fig. 3, there is no central server component maintaining the game state. All participating peers host both a client and a server component and are connected to each other in order to synchronize their local game state with each other every tick.

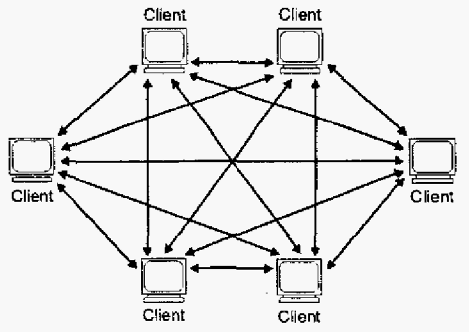


Fig. 3. Example of the peer-to-peer topology with six peers

There are two common ways to perform this synchronization. The first concept, which we call *state change propagation* in the following, is to update each of the peers' local states by processing the local user input. Each client then submits the local state changes to the other peers, which update their states accordingly. The other concept, referred to as *synchronized action processing*, only transfers the occurred user inputs between the game clients. Each client runs the full game state processing loop and performs all user actions, such that the game state at all peers is updated in the same way.

Both concepts have their assets and drawbacks. While the state change propagation method requires only little computing time, because only the local user actions are processed, it requires high communicational bandwidth to communicate state changes with the other peers. The concept of synchronized action processing requires much less bandwidth to communicate user actions which commonly are small in size; however, each client has to process all user actions, which requires much more computational resources than the state change propagation approach. Therefore, it depends on the game design which synchronization method to use. If the game state is reasonably small, i.e., each user only controls a single or a small number of avatars, and only small parts of the game world itself can be changed (typical for FPS games), the state change propagation should be used. If there are are a lot of avatars being controlled by users (typical for RTS games), then synchronized action processing is more suitable, as realized in *Age of Empires* [10], for example.

Both concepts require detecting and resolving inconsistencies of the distributed game state which result from concurrent user actions affecting the same part of the game state. A further discussion of this aspect can be found in [7].

In our scalability model, we only consider the peer-to-peer topology with state change propagation, because of two reasons: First, the GS model is intended to analyse the topologies' capability to support massively multiplayer sessions in the FPS and RTS genres. Although so far there are only few designs for such games, it is probable that the number of avatars controlled by each user is quite small, as is already the case for MMORPG games. Second, the proxy server topology uses a state change propagation synchronisation as well, which is discussed in Section II-F in detail. These topologies, therefore, can be compared more directly. Although we assume a smaller number of avatars controlled by each user in a massive-multiplayer RTS than in conventional RTS with only a few players, the partition of the game world as in MMORPG is still not possible. In RTS games, each player usually shares the view with his allied players in order to agree with these users on strategies and tactics for the game session. Therefore, the game client still would have to connect to a lot of different region servers to get informations about all the avatars of allied players.

In comparison to the client-server topology, a participating host in a peer-to-peer setup acts as a server processing the complete game world as well as a client processing user inputs and rendering the game visuals. Therefore, the length of a tick is not exclusively available for the game state processing by the server component, but has to be shared with calculations done by the client component. In *Age of Empires* [10] for example, the rendering of the game graphics requires 30 % of the available computing cycles at a peer. In our model, we therefore assume that not the complete length of a tick is available for the game world processing in a peer-to-peer session. The absolute amount of time required by the client component for the calculation of graphics, reading user inputs and playing sound effects depends on the actual implementation and differs from game to game. Advanced 3D engines, as commonly used in FPS games, are complex and could require a lot of the available computing cycles to render the visuals. Furthermore, for Internet-based sessions, there is usually less bandwidth available for a peer at a dial-up Internet connection than for dedicated servers hosted at game service providers with high-capacity Internet connections. We discuss these additional constraints in the scalability comparison in Section III.

For a peer, the main steps in the calculation of a single tick are as follows:

- Processing and validation of the actions of the local user, which takes a time of $t_{ca}(n, m)$ at maximum, depending on the total number of participating peers and simulated game entities.
- Receiving and processing of $n-1$ remote state updates, each of a constant size of d_{rsu} (remote state update), with a constant processing time of t_{rsu} for each message.
- General processing of the game world: Each client has to process m game-controlled entities, with a time of $t_{se}(n, m)$ for each entity in the worst case.
- Transmission of updated state information S_{i+1} to $n-1$ remote clients with an amount of data of d_{rsu}. Although the peer-to-peer topology would benefit a lot from the usage of multicast communication, our model assumes unicast communication for this transmission, because most of the Internet Service Providers do not support IP-multicast for home dial-up connections.

Therefore, the overall time for the calculation of a single tick at each client is

$$T_{p2p}(n,m) = t_{ca}(n,m) + (n-1) \cdot t_{rsu} + m \cdot t_{se}(n,m) \quad (4)$$

The amount of utilized bandwidth both for incoming and outgoing traffic at each client is

$$D^{in}_{p2p}(n) = (n-1) \cdot d_{rsu} = D^{out}_{p2p}(n) \quad (5)$$

F. Proxy Server Network Topology

The proxy server topology was sketched in [16], its enhancements and an implementation were described in [6]. In this multi-server network architecture, several so-called proxy servers are interconnected via multicast communication (IP- or application level). Each of the proxies has full view on the game state. Clients are connected to one of the proxy servers in order to participate in the game session. Fig. 4 depicts an example session in the proxy server topology.

The proxy server approach differs from multi-server topologies for MMORPG that usually segment the game state and distribute the authority for different parts of the state and world map exclusively to participating servers, which forces clients to change the server connection at runtime if the player moves into another region. The proxy server architecture does not require reconnects of clients at runtime because of the full replication of the game state and, therefore, is suitable for FPS and RTS games as well.

In general, the segmentation of the game world, as commonly done in MMORPG and described in [1], is not possible for FPS and RTS games. In team-based, massive multiplayer FPS games like *Battlefield 1942*, players have a huge viewing range, especially players that use sniper rifles or flying vehicles like planes or helicopters. Because of the segmentation, a server would only have access to state information about the region maintained by itself. Therefore, the clients would not the receive state information about adjacent regions, and the users could not be informed about game actions taking place in their range of view. In RTS games, players usually have control over more than one avatar which can be moved into different regions of the game world. The client would be forced to hold connections to several region servers. This problem is amplified by the fact that allied players in RTS games usually share their view, which further increases the number of server connections. Additionally, recurrent pauses in the commonly fast paced game play of both RTS and FPS games due to server reconnections because of the movement into a different partition would be annoying for users.

To provide an analytical model of scalability for the proxy server topology, we assume that each of the l proxies in a session has to serve k directly connected clients, which yields a total of $n = k \cdot l$ participating clients. We describe in the following the necessary steps in the calculation of a single state transition at a proxy server:

- Receiving, validating and processing of k actions from local clients, which requires the maximum time for processing $t_{ca}(n, m)$ for each client, depending on the total number of players and game entities. Each received message has a constant size of d_{cin}.
- Receiving and processing of $n-k$ remote client actions and state updates from other proxy servers. Each message has a constant size of d_{rsu}, with a constant processing time of t_{rsu} for each message.
- General processing of the game world: In the proxy server architecture, the administration of server-controlled entities is distributed among proxies, just like the authority for participating clients. With a distribution of s entities assigned to each of the l proxies such that $m = s \cdot l$, each server has to process s game-controlled entities, which requires a time of $t_{se}(n, m)$ for each entity in the worst case. Additionally, this requires each proxy to receive informations about $m-s$ remotely managed server entities, with a constant size of d_{rsu} for each message.
- Filtering and transmission of the new state S_{i+1} to clients. Each proxy has to filter the state information for its k clients, which takes $t_{fc}(n, m)$ for each client; each message sent has a size of $d_{cout}(n, m)$.
- Transmission of updated state information S_{i+1} to $l-1$ proxy servers. Information about the k local clients and

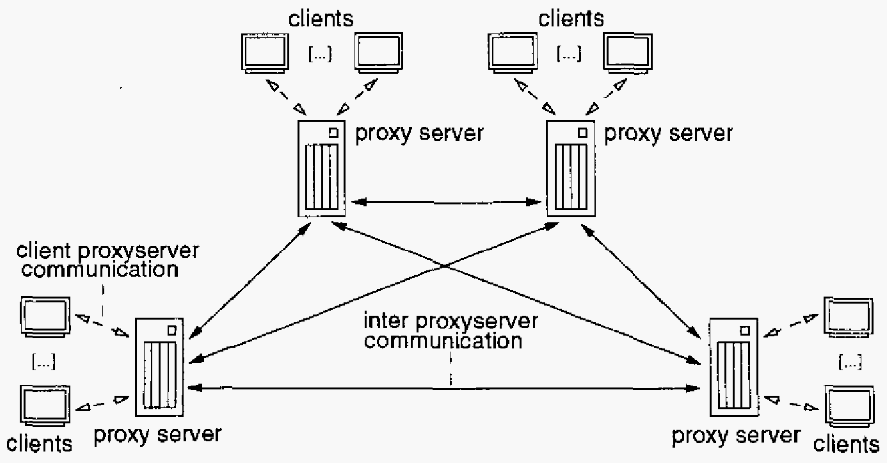


Fig. 4. Proxy server topology

the s server-controlled entities has to be sent, each with an amount of data of d_{rsu}. If IP-multicast is available between the proxies, this information has to be sent only once. In order to enable multicast, the proxies could be placed in a local area network or communicate over the MBONE [17] in an Internet-wide setting. If it is not possible to include a particular proxy in the multicast communication, the updated state information has to be sent via unicast. If $p \in \{1,..,l-1\}$ denotes the minimum number of proxies with which multicast-communication is possible, the updated state information has to be sent $l-p$ times.

The overall time for the calculation of a single tick at a proxy server results in

$$T_{prxy}(l,n,m) = \frac{n}{l} \cdot t_{ca}(n,m) + (n - \frac{n}{l}) \cdot t_{rsu} + \frac{m}{l} \cdot t_{se}(n,m) + \frac{n}{l} \cdot t_{fc}(n,m) \quad (6)$$

At each tick, a proxy server utilizes bandwidth for incoming traffic amounting to

$$D^{in}_{prxy}(l,n,m) = \frac{n}{l} \cdot d_{cin} + (n - \frac{n}{l}) \cdot d_{rsu} + (m - \frac{m}{l}) \cdot d_{rsu} \quad (7)$$

while outgoing traffic requires a bandwidth of

$$D^{out}_{prxy}(l,n,m) = \frac{n}{l} \cdot d_{cout}(n,m) + (l-p) \cdot (\frac{n}{l} + \frac{m}{l}) \cdot d_{rsu} \quad (8)$$

G. *Summary of the GS Model*

Tables I and II summarize the GS model by providing the calculation time and utilized bandwidth in the processing of a single tick for the three different topologies according to our model. For a specific game, the coefficients for computational time $t_{ca}(n,m)$, $t_{se}(n,m)$, t_{rsu} and $t_{fc}(n,m)$ can be calculated or measured as well as the coefficients for data sizes d_{cin}, d_{rsu} and $d_{cout}(n,m)$.

There are two ways to use the model in order to reflect the relation between the maximum number of users and the tickrate f of the game. The first possibility is to calculate the maximum number of possible users in an uncongested session. Consider tickrate f, the number of server entities m, the number of proxies l (if the proxy-server network is used) as well as the maximal bandwidth for incoming D^{in}_{max} and outgoing communication D^{out}_{max} of the processing hosts to be given. We denote the calculation time for a single tick in this case as $T_{top}(n)$ and the required bandwidth as $D^{out}_{top}(n)$ resp. $D^{out}_{top}(n)$ with top to be one of the three topologies cs, $p2p$ or $prxy$. Then $n_{max} = \{max\ n : T_{top}(n) \cdot f < T_{max} \wedge D^{in}_{top}(n) \cdot f < D^{in}_{max} \wedge D^{out}_{top}(n) \cdot f < D^{out}_{max}\}$ is the estimated maximum number of participating clients in an uncongested session of a certain game. T_{max} is the maximum computation time available for state calculations in a single second. For the client-server and proxy server approach $T_{max} = 1s$ applies, while in a peer-to-peer setup T_{max} is of [0..1] seconds with $1s - T_{max}$ being the calculation time of the client component on the same peer for rendering graphics and processing user inputs as discussed in II-E.

The second way to use the model's formulae is to calculate the maximum possible tickrate f and therefore the degree of game's responsiveness for a given number of users n. In this case, $f_{max} = \{max\ f : T_{top}(n) \cdot f < T_{max} \wedge D^{in}_{top}(n) \cdot f < D^{in}_{max} \wedge D^{out}_{top}(n) \cdot f < D^{out}_{max}\}$ is the maximum tickrate at which server components will not exceed the ticks' length during the calculation.

There is of course a general trade-off between maximum tickrate, i.e. game responsiveness, and the maximum number of users. However, without an analytical model as ours, the quantitative analysis of this relationship is only possible by an extensive post mortem analysis of a complete game implementation. As discussed above, the GS model provides the possibility for a quantitative, a priori estimation based on the measurements for the basic computational and communicational tasks. This way, game developers can evaluate the system design much earlier in the design process.

TABLE I

CALCULATION TIMES IN GAME PROCESSING AT A SINGLE TICK

	calculation time
$T_{cs}(n, m)$	$n \cdot t_{ca}(n, m) + m \cdot t_{se}(n, m) + n \cdot t_{fc}(n, m)$
$T_{p2p}(n, m)$	$t_{ca}(n, m) + (n - 1) \cdot t_{rsu} + m \cdot t_{se}(n, m)$
$T_{prxy}(l, n, m)$	$\frac{n}{l} \cdot t_{ca}(n, m) + (n - \frac{n}{l}) \cdot t_{rsu} + \frac{m}{l} \cdot t_{se}(n, m) + \frac{n}{l} \cdot t_{fc}(n, m)$

TABLE II

UTILIZED BANDWIDTH IN GAME PROCESSING AT A SINGLE TICK

	utilized bandwidth
$D^{in}_{cs}(n)$	$n \cdot d_{cin}$
$D^{out}_{cs}(n)$	$n \cdot d_{cout}(n, m)$
$D^{in}_{p2p}(n)$	$(n - 1) \cdot d_{rsu}$
$D^{out}_{p2p}(n)$	$(n - 1) \cdot d_{rsu}$
$D^{in}_{prxy}(l, n, m)$	$\frac{n}{l} \cdot d_{cin} + (n - \frac{n}{l}) \cdot d_{rsu} + (m - \frac{m}{l}) \cdot d_{rsu}$
$D^{out}_{prxy}(l, n, m)$	$\frac{n}{l} \cdot d_{cout}(n, m) + (l - p) \cdot (\frac{n}{l} + \frac{m}{l}) \cdot d_{rsu}$

III. SCALABILITY COMPARISON

In the following, we compare the three discussed network topologies with respect to their utilization of computational and communicational resources.

A. *Client-Server vs. Peer-to-Peer*

From (1) and (4), it follows that

$$T_{p2p}(n, m) < T_{cs}(n, m)$$
$$\Leftrightarrow (n - 1) \cdot t_{rsu} < (n - 1) \cdot t_{ca}(n, m) + n \cdot t_{fc}(n, m)$$

For large values of the number of participating clients n, this relation can be simplified to

$$T_{p2p}(n, m) < T_{cs}(n, m) \Leftrightarrow t_{rsu} < t_{ca}(n, m) + t_{fc}(n, m)$$

Therefore, the peer-to-peer architecture utilizes less computational resources for the calculation of a single tick than the client-server approach, because the values of $t_{ca}(n, m)$ and $t_{fc}(n, m)$ increase with a growing number of n while the time for state updates t_{rsu} is constant.

The comparison of utilized bandwidth for incoming communication yields

$$D^{in}_{p2p}(n) < D^{in}_{cs}(n) \Leftrightarrow (n - 1) \cdot d_{rsu} < n \cdot d_{cin}$$

Again, for large values of n this relation can be simplified to

$$D^{in}_{p2p}(n) < D^{in}_{cs}(n) \Leftrightarrow d_{rsu} < d_{cin}$$

A single message containing a user action of size d_{cin} usually consists of an opcode and parameters for the action and, therefore, is small in size. A single state update of size d_{rsu}, on the other hand, can be of various size, depending on the actual game design. However, in Section II-E we assumed that a single user action usually alters only a little part of the game state in MMG game designs. Therefore, the values of d_{cin} and d_{rsu} are of similar size, resulting in $D^{in}_{p2p}(n) \simeq D^{in}_{cs}(n)$.

The bandwidth comparison of outgoing traffic results in

$$D^{out}_{p2p}(n) < D^{out}_{cs}(n)$$
$$\Leftrightarrow (n - 1) \cdot d_{rsu} < n \cdot d_{cout}(n, m)$$

which, assuming large values of n, can be simplified to

$$D^{out}_{p2p}(n) < D^{out}_{cs}(n) \Leftrightarrow d_{rsu} < d_{cout}(n, m)$$

In the right inequality, the size of outgoing informations to clients in the client-server topology, $d_{cout}(n, m)$, depends on the total number of participating clients n and is supposed to grow with an increasing n, while the size for remote state updates d_{rsu} in the peer-to-peer topology is constant. The server component in a single peer only has to send the resulting state changes of the actions of a single user, while the server in the client-server approach transmits the state changes of all visible avatars to a particular client. Regardless of the number of avatars controlled by each user, we can assume $d_{rsu} \simeq d_{cout}(1, 0)$, because in both cases the same basic information like the position and viewing direction of a single avatar has to be transferred. Therefore, with a high number of users participating, $D^{out}_{p2p}(n)$ is much smaller than $D^{out}_{cs}(n)$.

Besides this plain model comparison, there are some constraints, especially in the peer-to-peer topology, which influence its suitability for certain games. While the client-server approach utilizes computational time and bandwidth at a **single** dedicated host which usually is connected to the Internet using a high bandwidth connection, in the peer-to-peer topology **all** hosts require resources at the rate described by the model. The available resources at the single server limit the number of players for a particular game in the client-server topology, while in the peer-to-peer approach the number of players is limited by the weakest host. Especially for games which require a high tickrate for a fluent game play like FPS games, users with low bandwidth dial-up Internet connections are not able to participate in a peer-to-peer game session with a high number of players.

B. *Client-Server vs. Proxy Server*

The comparison of (1) and (6) results in:

$$T_{cs}(n, m) > T_{prxy}(l, n, m) \Leftrightarrow$$
$$(n - \frac{n}{l}) \cdot t_{ca}(n, m) + (m - \frac{m}{l}) \cdot t_{se}(n, m) +$$
$$(n - \frac{n}{l}) \cdot t_{fc}(n, m) > (n - \frac{n}{l}) \cdot t_{rsu}$$

In the inequality on the right hand side of $\Leftrightarrow$, the sum $t_{ca}(n,m)+t_{se}(n,m)+t_{fc}(n,m)$ is greater than the constant time t_{rsu} for large enough n. Therefore, with an increasing number of proxies l, a single proxy requires far less time for the calculation of a tick than the single server, thus allowing more clients to participate in a game session using the proxy topology.

The comparison of required bandwidth for incoming traffic yields:

$$D_{cs}^{in}(n) > D_{prxy}^{in}(l,n,m)$$
$$\Leftrightarrow (n-\frac{n}{l})\cdot d_{cin} > (n+m-\frac{n}{l}-\frac{m}{l})\cdot d_{rsu}$$

If we assume $d_{cin}\simeq d_{rsu}$ as it has been done in the bandwidth comparison of the client-server and peer-to-peer in III-A, the relation can be simplified to

$$D_{cs}^{in}(n) \;>\; D_{prxy}^{in}(l,n,m) \;\Leftrightarrow\; 0 \;>\; (m-\frac{m}{l})\cdot d_{rsu}$$

The obtained formula tells us that a single server in the proxy topology constantly utilizes more bandwidth for incoming communications than the single server in the client-server approach. Each proxy, in addition, has to receive state updates of the distributed server entities. The total amount of additionally received data depends on the numbers of entities m and proxies l and, therefore, on the game design.

The bandwidth comparison of outgoing traffic leads to:

$$D_{cs}^{out}(n) > D_{prxy}^{out}(l,n,m)$$
$$\Leftrightarrow (n-\frac{n}{l})\cdot d_{cout}(n,m) > (l-p)\cdot(\frac{n}{l}+\frac{m}{l})\cdot d_{rsu}$$

With an increasing number of participating clients n, the single server of the client-server topology requires more bandwidth for outgoing communication than a proxy, because the value of $d_{cout}(n,m)$ increases with the growing n while d_{rsu} remains constant. If multicast is available for all or at least some proxies, i.e., state updates have to be sent only $l-p$ - times, the bandwidth utilization at a single proxy is reduced further.

C. Peer-to-Peer vs. Proxy Server

The subtraction of (4) from (6) results in:

$$\begin{aligned}&T_{p2p}(n,m) > T_{prxy}(l,n,m) \Leftrightarrow\\ &\quad(\frac{n}{l}-1)\cdot t_{rsu}+(m-\frac{m}{l})\cdot t_{se}(n,m) >\\ &\quad(\frac{n}{l}-1)\cdot t_{ca}(n,m)+\frac{n}{l}\cdot t_{fc}(n,m)\end{aligned}$$

With a high value of $(m-\frac{m}{l})\cdot t_{se}(n,m)$, i.e., a game with a lot of server entities m, the relation of $T_{p2p}(n,m)$ and $T_{prxy}(l,n,m)$ depends on the other values of $t_{ca}(n,m)$ and $t_{fc}(n,m)$, and therefore on the actual game design. If there are only a few server entities m in comparison to the number of clients n in the game, the peer-to-peer topology performs better than the proxy-server approach because t_{rsu} is constant. However, similarly to the client-server topology, the proxy network requires the computational resources at dedicated hosts, while the processing in the peer-to-peer topology is done at hosts which additionally have to process user inputs and render the game graphics.

The bandwidth comparison of incoming traffic results in

$$\begin{aligned}&D_{p2p}^{in}(n) > D_{prxy}^{in}(l,n,m)\\ \Leftrightarrow &(n-1)\cdot d_{rsu} > \frac{n}{l}\cdot d_{cin}+(n+m-\frac{n}{l}-\frac{m}{l})\cdot d_{rsu}\\ &\Leftrightarrow 0 > \frac{n}{l}\cdot d_{cin}+(1+m-\frac{n}{l}-\frac{m}{l})\cdot d_{rsu}\end{aligned}$$

A client in the peer-to-peer topology always requires less bandwidth for incoming communication than a single proxy in the proxy server approach. On the one hand, the server has to receive $\frac{n}{l}$ user inputs of size d_{cin}, while the inputs at a client in the peer-to-peer concept directly occur at the client's host. On the other hand, each peer processes all m server entities on its own, while the distribution of server entities among participating servers in the proxy topology requires receiving state updates of entities from other servers.

The comparison of required bandwidth for outgoing traffic yields:

$$\begin{aligned}&D_{p2p}^{out}(n) > D_{prxy}^{out}(l,n,m)\\ \Leftrightarrow &(n-1)\cdot d_{rsu} > \frac{n}{l}\cdot d_{cout}(n,m)+\\ &\qquad (l-p)\cdot(\frac{n}{l}+\frac{m}{l})\cdot d_{rsu}\end{aligned}$$

Although $d_{cout}(n,m)$ grows with the number of participating clients, the overall amount of outgoing traffic to clients, $\frac{n}{l}\cdot d_{cout}(n,m)$, can be reduced by increasing the number of proxies l. The relation of $D_{p2p}^{out}(n)$ and $D_{prxy}^{out}(l,n,m)$ therefore heavily depends on the game design and the number of participating proxies l.

D. Summary of Comparison

1) Analytical Topology Comparison: We have demonstrated how our scalability model allows to analytically compare the suitability of different network topologies for certain classes of massively multiplayer game designs, as envisaged in the introduction.

As shown in Section III-A, the peer-to-peer topology requires less computational and communicational resources than the single server in the client-server topology, and generally allows more users to participate. However, the available resources for peer-to-peer are far more restricted because the host machine has to render the visuals as well, and a lot of dial-up Internet connection classes provide only little bandwidth. For fast paced game designs which require a high tickrate like FPS, the client-server topology performs better than peer-to-peer because it allows users with a low bandwidth connection to participate in the game. However, for RTS games, whose game-play usually is slower and does not require such a high tickrate, the peer-to-peer topology can be used as well.

Our analysis showed that the proxy server topology outperforms the client-server approach and should be used for MMG sessions in the FPS genre. In a game running at a relatively low tickrate, e.g., in MMORPG, the peer-to-peer concept can perform better than the proxy approach, depending on the game design. However, we focused on MMG sessions in FPS and RTS games. For MMORPG, there already exist networking concepts [1], [18], which are more suitable than the three topologies discussed here.

2) Decision Guidance: In order to have a decision guidance for game developers on which topology scales better for a particular game design, the coefficient values for the data amounts can be estimated very early in the game development process. As soon as the core game design is clear, developers know what kind of messages have to be transferred and can estimate their size. To obtain accurate values for the different calculation times from experimental measurements, the server data structures and the game state update loop have to be implemented, which already requires a lot of design decisions to be made. However, to obtain an estimation of the calculation times before any implementation is made, at least the complexity of the operations can be determined based on the design of the game-specific data structures and algorithms. Additionally, the calculation times of already existing games with similar application design can provide an estimation of the calculation times.

3) Worst Case Consideration: As discussed in II-B, our model assumes a game in which the worst-case situation is that all avatars are visible to each other and perform an action. However, the model can be adapted to other expected worst case scenarios: For particular game world with, e.g., two equal spots where players tend to meet regularly, the worst case could be assumed to consist in two distinct groups of avatars in which only avatars in one of the groups are visible to each other. To adapt our model to this worst case definition, $d_{cout}(n, m)$ has to be substituted by $d_{cout}(\frac{n}{2}, m)$ and $t_{fc}(n, m)$ by $t_{fc}(\frac{n}{2}, m)$ because only half of the players are visible for a particular client. This way, the GSM itself is able to handle any deterministic worst case scenario. However, we are aware that the determination of the expected worst case can be complicated for a complex game. Therefore, several worst case scenarios should be checked for sophisticated game designs. Another possibility to handle complex game designs is to make the worst case nondeterministic by including probability distributions for the number of visible avatars or the number of interactions per tick. This way, the GS model can be used to calculate probabilities for game congestion based on different game situations. However, a discussion of this possible approach is beyond the scope of this paper.

IV. Experimental Verification

For the verification of our analytical model, we conducted several scalability experiments. We analyse a test game, forecast resource usage and a maximum number of possible players and compare this prediction to the results obtained by actually running sessions of the test game. Additionally, we discuss in detail how the model is instantiated for this particular game. Our third requirement to the GS model was the possibility to directly incorporate it into a particular game in order to allow a self-monitoring of resource utilization at runtime, which we demonstrate in the following.

A. Test Game

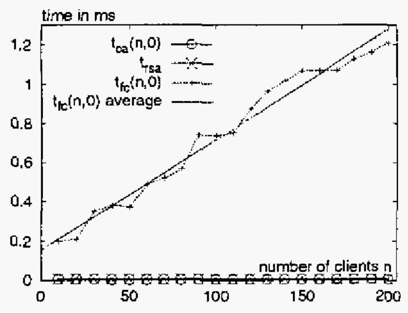
In [6], we already presented a test game for the proxy server-network, which we used again for the experimental verification of our analytical model and the demonstration of resource monitoring. In this game, each client has control over a single avatar. The game map, which is internally represented by a quadtree, consists of a bordered plane of constant size. Avatars walk around in the game map and try to catch avatars located nearby. Because there are no server entities in the game, the number of entities in the model is set to $m = 0$ for our experiments. Although this game is quite simple, the game state update loop is the same as in FPS games like *Half-Life* or *Unreal Tournament*. Additionally, we currently develop a sophisticated, massively multiplayer RTS game, which has a very similar communication concept.

In the experiments, the computational coefficients of the analytical model have been measured assuming the worst case for computational time, i.e., all avatars are visible to each other because each avatar's area of view covers the whole game map.

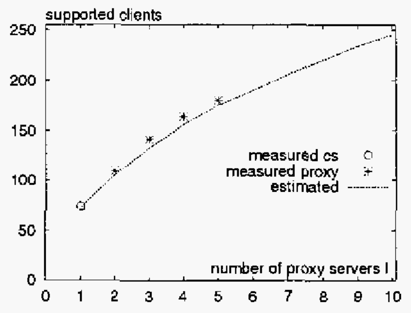
B. Determining Model Coefficients

Figure 5(a) depicts the measured values for the coefficients $t_{ca}(n, 0)$, t_{rsu}, $t_{fc}(n, 0)$ and an average graph for $t_{fc}(n, 0)$, which was calculated using linear regression and is used in the following model evaluation. We used hosts with a Pentium IV 2,6 GHz CPU and 1024 MB RAM running Linux with kernel 2.4.20 in a Local Area Network. We repeated all measurements several times, but there was hardly any variation of results because the game was constantly processing the worst case. At each tick, all avatars were visible to each other and every avatar performed an action. In our test game, $t_{ca}(n, 0)$ is independent of the number of players because client actions are simple moving or catching commands which only can affect a single opponent. Therefore, it has an approximately constant value of about 0.06 ms. The value of t_{rsu} is constant at about 0.0056 ms. Our implementation of the data filtering has a complexity of $O(n)$, due to which the function $t_{fc}(n, 0)$ grows linearly with an increasing number of players.

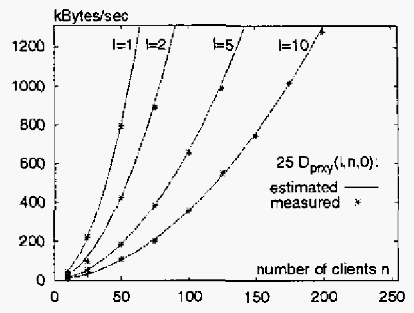
The communication coefficients were calculated by inspecting the message sizes sent between processes in our test game, resulting in $d_{cin} = 24\,\text{Bytes}$, $d_{rsu} = 21\,\text{Bytes}$ and $d_{cout}(n, 0) = 12\,\text{Bytes} \cdot n + 37\,\text{Bytes}$. This calculation includes protocol headers and acknowledgments for UDP messages. For the peer-to-peer topology, the substitution of thus obtained coefficient values into (5) yields $D_{p2p}^{in}(n) = (n-1) \cdot 21 Bytes = D_{p2p}^{out}(n)$ for a single tick. When using a tickrate of 25, as it is common for fast-paced FPS games, our model tells us that peer-to-peer can only handle six players at a 56 kBit half-duplex (e.g., modem) connection and 125 players at a 512 kBit full-duplex (cable or DSL) connection. In the client-server and proxy topologies, less data is sent to the clients since they do not have to process the complete game state update loop. According to our model, this allows the clients to handle information about 19 players at 56 kBit and 212 players at 512 kBit dial-up connections. Each client has to send user actions of size d_{cin} and to receive informations



(a) Computational coefficients



(b) Estimated and measured maximum player numbers



(c) Estimated and measured bandwidth utilization

Fig. 5. Experimental results for the test game at tickrate 25

of size $d_{cout}(n, 0)$ 25-times a second. The client in the peer-to-peer topology has to be connected via a symmetric cable or DSL link in order to support the 125 users at a full-duplex bandwidth of 512 kBit, because the bandwidth of $(n-1) \cdot d_{rsu}$ is required both for incoming and outgoing traffic each tick. A client in the client-server or proxy approach only requires a high bandwidth capacity of the connection's downstream to receive the game informations in height of $d_{cout}(n, 0)$ each tick. Therefore, an asymmetric DSL connection (e.g., 512 kBit downstream and 128 kBit upstream) can be used as well because the outgoing traffic only requires $25 * d_{cin} = 25 * 24 Bytes \simeq 4,7 kBit$ per second.

Additionally, the amount of data a client has to receive can be reduced by using a prioritization mechanism, which is not possible in the peer-to-peer approach because all state updates have to be received in order to ensure consistency of the game state. Therefore, it is difficult for the peer-to-peer topology to serve massive multiplayer sessions with a high tickrate because of the too high amount of required bandwidth. However, it requires only little computational time for the calculations in our test because there are no server entities to process and the processing of updates t_{rsu} requires only little time. Because of the present bandwidth limitation in the peer-to-peer approach, we concentrated on the client-server and proxy topology in our measurements of maximum player numbers.

C. Comparison of Estimation and Measurements

We measured the maximum number of participating clients in a non-congested session of our test game by adding clients to sessions running at a tickrate of 25 updates per second until the servers became congested. Servers indicate their congestion automatically when the calculation time for ϕ_{S_i,S_i+1} takes longer than 40 ms, the length of a tick. Figure 5(b) shows the maximum number of players estimated using the analytical model as well as the numbers actually measured. The maximum player number in the client-server topology is indicated by $\circ$, while the remaining measurement values indicate the maximum number of players in the proxy network with an increasing number of proxies l. As stated in the introduction, the number of hosts in our testbed limits the number of proxies to a maximum value of five servers in the test game, while we are able to forecast the player numbers using the model for an arbitrary number of proxies l as done for up to ten proxies in the plot.

For the evaluation of the model's forecast of bandwidth utilization, the bandwidth has been aggregated into a single value of $D_{prxy}(l, n, 0) = D^{in}_{prxy}(l, n, 0) + D^{out}_{prxy}(l, n, 0)$ for the sake of brevity. The curves in Figure 5(c) show the estimated amount of utilized bandwidth of $25 \cdot D_{prxy}(l, n, 0)$, while the dots denote the amount actually measured. The utilized bandwidth of client-server is represented by $D_{prxy}(1, n, 0) = D_{cs}(n)$. We observe that the estimation of utilized bandwidth is very accurate with a relative error not higher than 3 %.

The maximum number of players limited by the required computational time is consistently underestimated in our model at a rate of up to 7 %. This underestimation results from the inaccuracy of the `gettimeofday` function for short time interval measurements and from the usage of the averaged $t_{fc}(n, 0)$. However, with the underlying deterministic worst case, the GS model proved to be accurate enough to forecast the number of maximum players in a non-congested game at an adequate level and confirm the scalability of the proxy topology independently of the number of available test hosts.

D. Model Incorporation

To obtain the computation times of a running game instance, we included several measurement points into the already existing game source code. Since the main parts of calculation like action processing, update of server entities and filtering and sending of state updates to clients and other proxies already are encapsulated in appropriate functions, the time measurements could be easily incorporated into the code around the calls of these functions. Of course, we do not expect that all other game server implementations are that easy to adapt, but since the main computational steps are usually the core elements in the source code we assume that the incorporation of the time measurements into a game implementation is quite straightforward.

The communication coefficients can be determined quite easily as well. For this, the game developer needs to know what messages are sent in what situation and the absolute size of the transferred data. This should be easy to achieve by statically inspecting the source code, as we did for our test game.

Overall, the incorporation of the model for the resource monitoring at runtime is feasible. We are currently working ton including the model into a real RTS game based on the proxy server topology.

V. Related Work and Conclusion

There has been a lot of research in the area of scalable network topologies for MMORPG [1] and Distributed Virtual Environments (DVEs) [19]. However, the efficient usage of scalable networking for other genres of computer games has been rarely discussed. Our analytical scalability model is more detailed than previous comparisons [8] and allows an explicit scalability comparison of different topologies for a given game. The experimental verification demonstrated a good accuracy of the model's predictions. In [12], the resource usage of the single server of the FPS game *Quake* has been discussed in detail. However, the presented results and scalability optimizations are specific for this particular game and are difficult to generalize to other classes of multiplayer games or other network architectures.

Besides scalability, there are other important characteristics of networking topologies for computer games. These factors have to be taken into account when deciding what topology is most suitable for a given game design. In the peer-to-peer topology, cheating at a hacked game client (with the goal to reveal informations that should not be visible to the user) is possible, because the complete game state resides in the host's memory [15]. The server in the client-server topology forms a single point of failure for the complete session, while in the proxy approach only the clients directly connected to the failed proxy would be disconnected from the session. Furthermore, these clients could be reconnected to an other proxy in order to stay connected, because each proxy has the required state informations of these clients due to the full state replication.

For massive multiplayer games, scalability is usually more important than for conventional FPS or RTS game designs. We showed that the presented scalability model allows an analytical comparison of the three discussed network topologies and serves as a decision guidance what topology to use for a particular game design as summarized in Section III-D. There also exist other topologies not included in this comparison. By identifying the basic subtasks in the calculation and communication of the game state, which seems to be applicable to other communication concepts as well, our model is open for extension to towards other topologies. For example, Knuttson et. al *[18]* presented a peer-to-peer overlay which groups peers according to their area of interest and therefore lessens bandwidth utilization. Although this concept is more suitable for MMORPG than for FPS and RTS sessions, a comparison of this peer-to-peer overlay to other topologies based on our model is possible.

The test implementation shows that the model can be incorporated into games in order to monitor resource utilization and to forecast possible player numbers at runtime. In future work, we plan to use the model in order to implement an automatic load balancing mechanism in the proxy topology in order to efficiently support a high number of users in a heterogeneous, Internet-wide session setup.

Acknowledgment

We like to thank the anonymous reviewers, whose comments helped us a lot to improve this paper.

References

[1] Wentong Cai, Percival Xavier, Stephen J. Turner, and Bu-Sung Lee, "A scalable architecture for supporting interactive games on the internet," in *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, Washington, D.C., May 2002, IEEE, pp. 60–67.

[2] Valve, "Counterstrike" <http://www.counter-strike.net/>, 1999.

[3] Codemasters, "Operation Flashpoint" <http://www.codemasters.com/flashpoint/>, 2001.

[4] Electronic Arts, "Battlefield 1942" <http://www.battlefield1942.ea.com/>, 2002.

[5] Ensemble Studios, "Age of Empires" <http://www.microsoft.com/games/empires/>, 1997.

[6] Jens Müller, Stefan Fischer, Sergei Gorlatch, and Martin Mauve, "A Proxy Server-Network for Real-time Computer Games," in *Euro-Par 2004 Parallel Processing*, Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, Eds., Pisa, Italy, August 2004, vol. 3149 of *Lecture Notes in Computer Science*, pp. 606–613, Springer-Verlag.

[7] Joseph D. Pellegrino and Constantinos Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," in *NetGames 2003 Proceedings*, May 2003.

[8] Jouni Smed, Timo Kaukoranta, and Harri Hakonen, "Aspects of networking in multiplayer computer games," in *Proceedings of International Conference on Application and Development of Computer Games in the 21st Century*, Hong Kong SAR, China, November 2001, pp. 74–81.

[9] Tim Sweeney, "Unreal networking architecture <http://unreal.epicgames.com/network.htm>," July 1999.

[10] Paul Bettner and Mark Terrano, "1500 archers on a 28.8: Network programming in Age of Empires and beyond," in *Game Developer Conference Proceedings*, San Jose, California, USA, March 2001.

[11] Lars C. Wolf Lothar Pantel, "On the suitability of dead reckoning schemes for games," in *NetGames 2002 Proceedings*, April 2002.

[12] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos, "Behavior and performance of interactive multi-player game servers," in *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, Arizona, USA, November 2001.

[13] Ahmed Abdelkhalek and Angelos Bilas, "Parallelization and performance of interactive multiplayer game servers," in *8th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 2004.

[14] Yahn W. Bernier, "Latency compensating methods in client/server in-game protocol design and optimization," in *Proceedings of Game Developers Conference'01*, 2001.

[15] Matt Pritchard, "How to hurt the hackers: The scoop on internet cheating and how you can combat it," *Game Developer*, June 2000.

[16] Stefan Fischer, Martin Mauve, and Joerg Widmer, "A generic proxy system for networked computer games," in *NetGames 2002 Proceedings*, April 2002.

[17] Hans Eriksson, "Mbone: The multicast backbone," *Communications of the ACM*, pp. 56–60, August 1994.

[18] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins, "Peer-to-peer support for massively multiplayer games," in *IEEE Infocom*, March 2004.

[19] E. Frecon and M. Stenius, "DIVE: A scalable network architecture for distributed virtual environments," *Distributed Systems Engineering Journal*, 1998.