Latency-sensitive, persistent-state distributed applications on P2P architectures: A survey

James Walker
Department of Computer Science,
Michigan Technological University
jwwalker@mtu.edu

November 8, 2011

Abstract

I investigate the current state of research regarding the deployment of latency-sensitive, persistent-state distributed applications (LPAs) on peer-to-peer architectures. P2P systems present many unique challenges to LPAs due to their requirements for rapid response to user actions and a persistent global state despite the lack of centralized storage and computing resources. After establishing the reasons for the prevalence of client-server architectures and motivating a desire to enable deployment on P2P systems, I enumerate the requirements of LPAs and explore the challenges and solutions for meeting these requirements.

1 Introduction

Latency-sensitive, persistent-state distributed applications (hereafter referred to as LPAs) are networked applications with heavy user interaction and a consistent global state that persists between logins, often with a graphically intensive visual component. LPAs are traditionally deployed on client-server architectures. While this approach has proven to be effective, there are valid reasons why developers may prefer to use a P2P architecture. Due to the significant technical challenges of deploying LPAs on P2P systems, there are virtually no examples of large-scale projects using this technology. Enabling LPAs on P2P networks is therefore an active area of research.

The most prominent contemporary example of LPAs is massively multiplayer online gaming (MMOG), which has led to a rich body of research. MMOGs have become a billion-dollar industry [1], and the popular World of Warcraft MMOG alone has over 12 million players as of October 2010 [2], demonstrating the growing prevalence of this entertainment venue. Additionally, games are demanding applications that serve as an effective testing ground for cutting-edge techniques, and as such, the related research has serious practical applications.

This paper is structured as follows. First, I establish the prevalence of client-server architectures in MMOGs and explore the reasons for this prevalence. Next, I establish
reasons for considering a P2P over a client-server model for use by LPAs. Finally, I explore the most significant challenges and proposed solutions for accomplishing this migration.

2 Prevalence of the client-server model

Virtually all modern MMOGs, whether commercial or freeware, run on a client-server architecture. A representative (but by no means comprehensive) example of successful MMOGs that use a client-server architecture includes World of Warcraft [3], Rift [4], EVE Online [5], Lord of the Rings Online [6], Dungeons & Dragons Online [7], and Mabinogi: Fantasy Life [8]. I could find no examples of mature MMOGs built on a P2P architecture.

This prevalence suggests compelling reasons for using a client-server architecture. The following reasons are particularly relevant:

Ease Fan et al [9] note that it is relatively easy to implement and secure MMOGs on a client-server architecture (compared to other approaches).

Maturity Techniques for coding client-server architectures are mature and reliable.

Scalability If a server gets overwhelmed, the developer can use the proceeds from the high subscription volume to purchase additional servers or upgrade existing servers.

Centralization Storing important application data on the server avoids problems with inconsistent data among the clients.

Security Performing important calculations on the server limits opportunities for cheating and malicious client behavior.

Finances By forcing the software to communicate with a central server, the developer can enforce subscription fees.

3 Reasons to use a P2P model

Despite the compelling reasons to use a client-server architecture, there are valid reasons to consider developing an LPA using P2P technology instead. This will become particularly true as the use of P2P models for LPAs matures, hence the rich body of research occurring in this field. Reasons to consider deploying an LPA on a P2P architecture include the following:

Hardware Supporting popular LPAs with the client-server model requires the use of expensive server hardware. Since a P2P model utilizes the computing power of the peers, significant cost savings could be realized.
Reliability  Servers represent a single point of failure. If a critical server goes down, users are unable to use the software until functionality is restored. A P2P network has no single point of failure and is thus more robust. Users could be guaranteed almost perpetual uptime.

Computation  As discussed by Douceur et al [10], an acknowledged weakness of the client-server model is the simplistic behavior of AI-controlled agents. This weakness extends to any computationally expensive task with substantial replication. A P2P network represents an opportunity to distribute complex computation among many peers, enabling more sophisticated behavior.

Scalability  Properly leveraged, a P2P network provides better scalability than a client-server model, because available resources grow in proportion to the number of users (Hampel et al [11]). This amounts to free maintenance, as contrasted with the need to purchase servers to accommodate a growing user base in the client-server model.

Aside from a pure P2P architecture, techniques for using a hybrid server-P2P model are also being actively researched. Various technical difficulties such as persistent storage of state data, as well as the issue of enforcing user subscriptions, could be resolved with this hybrid approach. In this model, the server performs dramatically reduced functions compared to a pure client-server model, but is still critical to the operation of the software. This idea has been considered by Chan et al [12] and Cronin et al [13].

4 Challenges and solutions

Having motivated the consideration of deploying LPAs on a P2P network, I examine the most significant challenges and proposed solutions. LPAs such as MMOGs have more sophisticated requirements than other P2P applications (Schiele et al [14]). Research into solving these daunting challenges is ongoing.

This section is divided into 10 subsections discussing the requirements of LPAs. For selected requirements, I discuss work that has been done in that area, including specific proposed algorithmic solutions. Other requirements are given only a general treatment.

4.1 Distribution

Large volumes of state data and potentially intensive computations must be effectively distributed among peers. It is crucial to avoid overburdening peers while ensuring that necessary work is performed in a timely and reliable manner. [14] In MMOGs, one of the greatest challenges is computing the behavior of AI-controlled agents, known as Non-Player Characters (NPCs). [9]

Douceur et al [10] discuss a technique for using distributed computing to improve these computations. Their work especially focuses on creating superior NPC behavior, but it can be generalized to any intensive computation that is replicated many times within the application.
The AI controlling computer agents in LPAs is very poor because the servers have insufficient computational power to support the demands of thousands of sophisticated concurrent agents. The algorithm discussed by the authors is designed to supplement a client-server architecture by offloading certain computations onto clients. It is therefore a hybrid approach rather than a pure P2P technique.

Issues that must be considered in client offloading include the availability of client CPU capacity, possibility of client failure, communication latency, and risk of exploitation. The system requirements of most LPAs compared to the average specifications of the computers running these applications suggests that most clients have CPU time to spare, but the other issues are less straightforward.

The central idea of the algorithm is to split the AI into two components, a server side and client side. The server side performs simple, tunable, high-frequency computations; while the client side computes tuning parameters for the server. The server sends “glimpses” of the application state containing only immediately relevant data to the clients and asks them to perform computations. The output is returned in the form of “advice” to the server-side AI, typically parameters or coefficients.

Because the server can’t guarantee that it will receive prompt advice, the results of client computations should be useful over a period of time and not expire quickly. The server should also have a fallback operation in case advice is not returned in time. Client computations should be stateless; that is, independent of their preceding and succeeding computations. This characteristic supports redundancy and makes exploitation more difficult. Finally, the client computations should be deterministic, which increases reliability (the server can distribute the same computation to multiple clients) and security (clients can vote on the result of computation). If randomness is required, this can be worked around by having the server supply the seed value.

As an example of this technique, the authors provide a specific algorithm for tactical navigation. Classic navigation generally consists of choosing a goal and then moving toward the goal via precomputed waypoints. When the agent comes within a certain range of its goal, it can switch to simple movement mechanisms such as approaching or circling. The authors propose an improvement in the form of aggregate influence fields.

The goal is to calculate a vector field that incorporates the influence of multiple concurrent priorities; for example, “target A is strongly attractive and target B is weakly repulsive.” This field optimizes over multiple goals and can be recalculated to accommodate changing circumstances. The navigator simply follows the path calculated by the field.

On the server side, the computation is approximated by a second-order two-dimensional Taylor series (omitted for reasons of space), and the clients are used to compute the correct coefficients. The approximation is accurate for the area immediately surrounding the navigator, but it can become wildly inaccurate around the edges of the field. Therefore, the field needs to be recomputed periodically.

To evaluate the algorithm, the authors ran an experiment on a modified Quake III engine. The experiment demonstrated that AI-controlled “bots” using the aggregate influence fields outperformed bots using the standard AI even with latency as high as 1 second. However, performance rapidly decreased beyond latency of 1 second. These results demonstrate the viability of the technique and also suggest 1 second as a
reasonable cutoff point for when the server-side AI should use its fallback calculations.

4.2 Consistency

An identical state must be preserved between all users. The ideal is that all users always perceive the effects of an event identically and simultaneously, but this ideal is not achievable in practice. [14] This requirement is also known as “interest management.” [9] [15]

Key observations are that it is unnecessary for each user to know about the entire application state, and users’ avatars generally have tightly limited senses and mobility. These observations can be exploited to make this problem more manageable. [9] [16]

4.3 Persistence

Changes made to the application’s common state must persist between users’ login sessions. When a user logs back in, all changes since the user’s last login should be immediately visible. This is one of the greatest challenges to implementing LPAs on P2P architectures due to the lack of centralized storage. [9]

UC Berkley developed a project to address this issue known as OceanStore [17], but more work has been done using Pastry. Hampel et al [11] and Knutsson et al [18] built applications specifically for this purpose on the Pastry architecture.

Pastry is a distributed hash table (DHT), enabling hash-like data access on a distributed network. The key-value pairs are stored in a redundant P2P network, with a dynamically built and maintained routing table. With no single point of failure, individual nodes can leave the network with minimal chance of data loss. A routing overlay network is built on top of the DHT for scalability and fault tolerance. [19] [20]

In Hampel et al’s [11] and Knutsson et al’s [18] work, Pastry is combined with PAST and Scribe to enable persistent state on a P2P network. PAST is a distributed file system layered over Pastry in which files are stored by computing the hash of their filenames [21] [22], while Scribe is a decentralized publication and subscription system that uses Pastry for route management and host lookup [23] [24]. First, objects are located using Pastry, then access is synchronized with Scribe, and replica management is handled by PAST.

This architecture technically fulfills the persistence requirements of LPAs. There is some question of whether this architecture is efficient enough to cope with the rigorous real-time requirements of an LPA, but in Knuttson et al’s work, a simulation with 4000 nodes showed an average message delay of 150ms, with per-peer bandwidth requirements averaging 7.2KB/sec and peaking at 22.34KB/sec. These results suggest that this architecture might be sufficiently scalable to be suitable for its intended purpose.

4.4 Availability

The application must be highly fault tolerant, provide multi-client support, and have minimal downtime. [14] This requirement could be somewhat relaxed in a freeware application, but paying customers expect the service to have over 99% uptime. Fortunately, high availability is an inherent characteristic of P2P networks.
4.5 Interactivity

Users’ actions must take effect with minimal delay and be visible to other users quickly. [14] This issue is addressed by Pang et al [15] in an algorithm that also deals with scalability, and is therefore discussed fully in that subsection.

4.6 Scalability

LPAs must be able to support thousands, tens, or even (in extreme cases) hundreds of thousands of simultaneous users. [14] P2P networks are often said to have superior scalability to client-server architectures, but this is true only when the strengths of the P2P model are properly leveraged. Every peer cannot simply send every message to every other peer, otherwise the network would be overwhelmed and performance would degrade dramatically. Therefore, methods for partitioning users are necessary.

Pang et al [15] propose an algorithm for handling this as well as the problem of prompt response to user interaction. They note that area of interest (AOI) filtering alone is not sufficient, because users often congregate in large numbers in a constrained area.

In the authors’ algorithm, each peer is made responsible for a subset of objects. The peer runs “primaries” of these objects, and other interested peers maintain a “replica” of that object which receives updates from the primary. Aside from this core concept, the authors identify 3 principles which inform their algorithmic design:

1. Users have bounded attention. A human can only focus on a constant number of objects at the same time. Thus, the total amount of attention in the application grows linearly with the number of users, not quadratically. This principle is addressed by focus sets.

2. Interaction must be timely and consistent, so inter-object writes should have the highest priority over other kinds of updates. This principle is addressed by pairwise rapid agreement.

3. Realism should not be sacrificed for accuracy. Users are more satisfied if objects obey the physics and rules of the application than if objects exhibit strange behavior to maximize consistency with their respective primaries. This principle is addressed by guidable AI.

Peers are organized into focus sets. A peer in a focus set sends updates to the peer focusing on it every frame, while peers not in the focus set only send best-effort updates based on available bandwidth. Peers are organized into sets based on the metrics of proximity, aim, and interaction recency. The exact formulas providing these metrics are omitted for reasons of space.

In pairwise rapid agreement (PRA), peers are differentiated as writers W, which modify targets T. Upon interaction, W and T communicate immediately so that they can quickly agree on the state of the modified object. T applies the write to its primary copy of the target and replies with the new state. W’s replica of the target may be in a state inconsistent with T’s primary, in which case the write is impermissible and W is notified that the change did not occur as intended. PRA is feasible because these
interactions occur on a human time scale, so they are not nearly as frequent as regular
updates. Replicas whose peer is not in the focus set of the primary receive infrequent up-
dates. By the standard dead-reckoning solution, mobile objects can exhibit “choppy”
movement, which users find unsatisfying. With guidable AI, the primary instead sends
its replica “guidance” about how it should act until its next update. The guidance con-
sists of 2 elements, prediction and action counter. The prediction is what the replica
expects its state to be in t seconds based on its current action, while the action counter
tells it how many times to perform a given action if it was currently performing a re-
peated action (e.g., weapons fire) during its last update.

This simulation of the primary’s behavior ensures smooth and realistic movement
while coarsely approximating the primary’s state. However, high inaccuracy is possi-
ble, so guidable AI is only used for peers not in the focus set. It is also possible, if it
enters the focus set, that the replica’s simulated state and the primary’s true state may
be far apart. In this case, the primary continues guiding the replica back to its true state
until they converge in order to avoid a jarring “teleportation” directly to the true state.

A study was conducted with 88 users which found that user satisfaction rises dra-
matically when using this algorithm on a low-bandwidth network. In fact, user satisfac-
tion was almost as high as when playing on a high-bandwidth network. These findings
suggest that the algorithm can be used to ensure timely responses to user interaction
and scale distributed applications by decreasing bandwidth usage.

4.7 Security

Users must have secure accounts and safe payment options. The application design
should also make cheating difficult and enable the investigation of virtual crime, such
as violations of the application’s EULA.[14]

A discussion of account and payment security is omitted, but I shall consider the
issue of cheating as addressed by Baughman et al.[16]. The authors show that exist-
ing synchronization protocols enable straightforward cheats. Worse, in some of these
paradigms, it is impossible in general to distinguish between fair play and cheating
behavior, and the protocols may even corrupt the global state, rendering interaction
meaningless.

The authors propose a superior synchronization/anti-cheating protocol in 2 parts.
The first part is a secure but inefficient algorithm called Lockstep. The second algo-
rithm, Asynchronous Synchronization, improves efficiency without sacrificing any of
the security of Lockstep.

Lockstep is a stop-and-wait protocol in which each process stops and waits for a
decision from every other process at each application frame. A commitment step is
added to prevent processes from waiting and analyzing other decisions to inform their
strategy (the “lookahead” cheat). Once a frame is complete, each process announces
a cryptographically secure one-way hash of its decision for the next frame. Once all
processes have committed, their decisions are revealed in plaintext. Thus each player
only has information on the current frame for making their next decision.

However, Lockstep always runs at the speed of the slowest peer. To improve per-
formance, the authors propose the Asynchronous Synchronization (AS) algorithm. In
AS, the application’s clock is decentralized and each process advances asynchronously from the others until interaction is required. Therefore, processes can advance in application time without contact from other processes.

Each process is assigned a sphere of influence (SOI) which conceptually represents an area around the user in which his actions could influence other users’ decisions. If two users’ SOIs don’t intersect, it is impossible for their actions to influence each other’s decisions, and it is unnecessary to enforce the Lockstep restrictions. Because processes advance their time frames independently, the SOI is not static. It is composed of 2 parts, the “base” plus a “delta,” where the delta is the change in the SOI that could occur in subsequent frames. The total of base + delta is used when computing intersections.

In AS, each process determines its decision for the current time frame, then announces the commitment of that decision to all other processes. Commitments that are one frame past the last revealed frame of another process are accepted. The local process determines which remote processes it is waiting for by computing SOI intersections. If no remote processes are in the wait state, the local process reveals its state, updates its known data about other processes, and advances to the next frame. There is no need to contact every process at every frame, only processes with an SOI intersection, which provides a significant performance advantage. The worst case occurs when a peer with maximum latency approaches a peer with maximum delta, then the slower peer must “catch up” to the faster one.

The primary flaw of this protocol is that each peer must be aware of all other peers’ locations at all times, which does not scale well. In order to make the protocol scalable to a massively distributed environment, the authors propose combining AS with the commonly employed cell-based approach. With this technique, users are clustered into separate multicast addresses based on position, so the virtual space is broken into “cells” which are transparent to users. By combining AS with cells, the protocol only needs to be invoked for other users inside the current cell. Lastly, the authors propose a cryptographic solution to prevent users from learning each other’s positions based on the necessary exchange of positional data.

4.8 Efficiency

Many LPAs consume substantial CPU resources. Furthermore, as discussed in the Distribution subsection, an effective P2P LPA should leverage the computational power of its peers, thus further burdening users’ CPUs. It follows that the computational cost of maintaining the application’s networking requirements should be minimized. Bandwidth utilization should also be as efficient as possible. [14]

Douceur et al [25] investigate both aspects of this problem. In the interest of utilizing as much upload bandwidth as possible in a P2P environment, they investigate the “latency-constrained upload bandwidth maximization problem” (MUBP). Their proposed algorithm is very complex, so details are omitted from this discussion. To summarize, the authors formalize MUBP as a combinatorial maximization problem, and show it to be NP-complete by a reduction to the 3-partition problem. They also note that algorithms with good average-case but poor worst-case behavior are unacceptable because in MMOGs, the worst-case scenario occurs often in practice (whenever large
numbers of peers focus on a single peer). They then present a worst-case efficient approximation algorithm whose total upload is close to the optimum even in worst-case scenarios.

4.9 Maintainability

When the software is updated, the update must be applied to all users. Although simpler applications may allow users to connect with different versions of the software without ill effect, in an LPA this scenario would seriously hinder meaningful interaction. This problem is trivial in a client-server architecture because the server can force users to download the latest version of the software before logging in. The problem is far more difficult on a P2P architecture because there is no centralized point of distribution. [14] I was unable to find any research devoted specifically to this problem.

4.10 Incentive Mechanisms

P2P is a voluntary resource sharing mechanism that relies on the cooperation of participants. Therefore, peers must be induced to share their resources. Two major approaches include accounting, in which a peer’s contributions are monitored and it is permitted to consume roughly equivalent resources; and reputation, in which a peer’s dependability and honesty are quantified based on actions such as abrupt disconnects [9]. This problem is less technically challenging than most of the others discussed in this paper, and so is not given further consideration.

5 Conclusion

At present, LPAs are deployed almost exclusively on client-server architectures. There are several reasons why client-server architectures are a sound choice, but many of those reasons simply boil down to the comparative complexity of P2P architectures. Many issues that are straightforward to implement on a client-server architecture, such as persistent state and global updates, present daunting obstacles on a P2P system. However, as the reliability and maturity of LPA-specific P2P architectures improves, significant resource savings could be realized by preferring these over the client-server model, due to the resources contributed by the peers themselves. The development of middleware incorporating proven techniques, or the standardization and dissemination of those techniques, would be a major step in the proliferation of P2P-driven LPAs.

References


