Effects of Network Latency on Games with Human and Distributed Agent Players

William Birmingham  Britton Wolfe
Computer Science Department, Grove City College
birminghamwp@gcc.edu, wolfebd@gcc.edu

Abstract
We are interested in mixed human and agent systems in the context of networked computer games. These games require a fully distributed computer system. State changes must be transmitted by network messages subject to possibly significant latency. The system then is composed of agents' mutually inconsistent views of the world state that cannot be reconciled because no single agent's state is naturally more correct than another's. The paper discusses the implications of this inconsistency for distributed AI systems. While our example is computer games, we argue the implications affect a much larger class of human/AI problems.

Introduction
We are interested in developing distributed AI (DAI) systems for multiplayer, networked video games. Networked video games are an excellent application for DAI, as they provide a mixed human/agent environment that must operate under real-time constraints. DAI agents can play various roles in these systems, such as enemies or (player) companions. For many game types, such as first-person shooters (FPS), the DAI agents must coordinate their actions to present the player with compelling game play.

Putting aside the formidable challenge of creating DAI agents that can “play” a game well, even relatively simple coordination tasks are daunting in a networked environment. Players, especially highly skilled ones, are sensitive to speed of interaction with other players and DAI agents; players dislike so-called “laggy” game sessions. Thus, both computational and network delays can significantly reduce the quality of game play. For example, network delays of 200 ms, not uncommon on the Internet, can negatively impact players of FPS games. (Claypool and Claypool 2006)

Developing DAI for these games presents multifaceted challenges, most stemming from the computational platform: the DAI agents must run on fully distributed computers. In other words, the machines share no memory, no common clock, and must exchange all state, including any synchronization signals, via network message passing. The network necessarily has the properties of latency and data (commonly, packet) loss. Distributed state and latency confound coordination algorithms that at least implicitly assume common memory or latency-free network transport. (Birmingham, et al., 2012)

While this paper uses networked computer games to illustrate the problems of latency and its effects on various DAI models, we note that the problems apply across a wide range of problems, particularly those where agents interact with humans. Generally, when decision-making time is less than latency and computation time, the issues described in this paper may appear.

Networked Games as the Pursuit Problem
We model games as a collection of agents composed of player agents (avatars) and non-player controlled (NPC) agents. Avatars are controlled completely by a human; NPCs are controlled completely by AI, and typically contain methods for tracking, navigating, playing the game (e.g., selecting tactics), and possibly coordinating or competing with other agents and the human players.

In networked games, each networked computer has its own agent, either NPC or avatar. When an agent needs to communicate state to another agent, such as updating its position or sending a coordinating signal, it sends a message. Typically, the messages are sent via UDP or TCP packets.

Copyright © 2017, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

1 For this paper, we treat packet loss as latency. TCP/IP handles packet loss through retransmission resulting in an increase in latency.

2 We note in many networked games, NPCs and avatars run on the same machine. We describe the general case: each agent runs on its own machine.
over the Internet. Because messages are sent over a real network, there will be latency. Consequently, the state that each computer maintains is necessarily out-of-date with respect to all other devices, since it is receiving a message (state) that is already stale from the perspective of the sender.

Before returning to the problem of stale data, we describe network games as a pursuit problem. We consider games where each agent resides on a different machine, and gameplay requires either cooperative or competitive behaviors among agents. The pursuit problem is a reasonable model for these types of game AI, because many of the NPC behaviors we design into our agents can be modeled, at least at a simple level, as a pursuit problem. We define the pursuit problem as the following set of agents:

- A set of predators $p_i \in P$ that attempt to capture the prey.
- A set of prey $r_j \in R$ that attempt to avoid being captured.

The predators and prey are characterized by many factors depending on the type of pursuit problem, but typically include the following attributes:

- velocity, $v$
- location, $l$
- movement algorithm, $m$, that normally considers positions of other agents,
- decision time, $t$.

For our research, the most critical attributes of the problem are $l$, $v$, and $t$.

In previous work, we showed that latency has deleterious effect on pursuit problems because of staleness. (Birmingham, et al. 2012) To illustrate, consider Figure 1 that shows the sequence of prey movement over three frames with network latency of one frame. The two rows illustrate the prey’s view of itself (top row) and the predator’s view of the prey (bottom row). Red circles show each agent’s view of the prey’s $l$ and open circles show the prey’s previous position(s). Notice that the prey’s and predator’s states are both correct yet inconsistent: there is no way to make the states entirely consistent without stopping the movement of the prey.

Generally, each agent maintains what it believes to an accurate state, which includes its knowledge of the positions of all other agents. Indeed, each state is accurate from the agent’s perspective; yet, there can be significant inconsistencies across agent states. This means a system-wide consistent state can never be guaranteed except in special cases. Network researchers have documented this problem for distributed (networked) game systems. (Armitage, Claypool, & Branch, 2006) Its manifestation in the pursuit problem, and generally DAI, is our concern.

### Implications

Because of latency and no common memory, each agent’s view of the world is correct, yet (likely) inconsistent with every other agent’s world model. Further, we cannot assume a latency-free clock (or synchronizing) signal. Thus, we require a system model that accounts for these things.

Common-memory models and models that assume no network latency, or where latency is not an issue, will not be guaranteed to work. Using Korf’s work (Korf 1992) as an example, he assumes that all agents know (minimally) $l$ instantly, because of common memory among agents (e.g., the algorithm runs on a single machine). Because we operate under real network conditions, agents have a stale view of each other’s $l$ and $v$.

Latency becomes a problem when the system of agents operates under real-time constraints. Roughly speaking, when $t$ (e.g., the time it takes to calculate the position to which to move) is less than latency time, the quality of the decision is imperiled. For example, most modern games maintain a framerate of 60 frames per second, which is approximately 17 ms; framerate is a good approximation of decision time for an NPC. With realistic, yet moderate, latency of 150 ms, nearly nine frames will elapse between a prey sending its position and the predators receiving it. Latency time far outstrips decision time.

There are several variations of partially observable Markov decision processes (POMDPs) intended for multiagent systems. These models appear applicable since they explicitly model the uncertainty in $l$ and, for a class of POMDPs called decentralized, do not assume common memory.

In these models, time proceeds in discrete steps. At each time step, each agent chooses an action and then receives some observation about the world. The model that appears to be a natural fit for distributed games is the decentralized partially observable Markov decision process (Dec-POMDP) (Bernstein et al., 2002). Like a POMDP, a Dec-POMDP includes a set of states $S$, a set of actions $A$, a transition function ($S \times A \rightarrow P(S)$) that yields a distribution over
next states, a reward function \((S \times A \rightarrow \mathbb{R})\), a set of observations \(O\), and an observation function \((A \times S \rightarrow P(O))\) that yields a distribution over observations. The Dec-POMDP extends a POMDP by adding a set of agents and imposing a structure on \(A\) and \(O\). The (joint) action set \(A\) is the Cartesian product of the actions \(A\), available to each agent \(i\). Similarly, \(O = X \times O_i\), where \(O_i\) is the set of observations for agent \(i\). Thus, each agent only selects part of the joint action and only observes part of the joint observation vector. In the special case where the joint observation at each time step reveals the state, one can model the system with a Dec-MDP (e.g., Roth, Simmons, & Veloso, 2007). In a Dec-MDP, the system can still be partially observable from any single agent's point of view. If the agents could instantaneously share their observations, they could see the state of the world. This instant communication, which is effectively shared observations, is assumed by the multiagent POMDP (MPOMDP) model (e.g., Messias, Spaan, & Lima, 2011). We note that observations are networked messages, and actions may also require networked messages. Thus, observation comes with the price of latency.

All of these POMDP-based models rely on the notion of state variables. In the pursuit problem, the state variables are the locations of the prey and each predator. These POMDP-based models assume there is one unique value for each state variable at each time point. However, in the environments we consider, agents exchange messages about the state variables with each other. Each message consists of a state variable and a value for that variable. All of the agents agree on the set of state variables and their semantics. For example, when \(p_3\) broadcasts its location, the other agents know how to update their views of the world.

While agents agree on the set of state variables, different network conditions across time and between different pairs of agents mean that each agent has a potentially conflicting view of the current values of those state variables. An agent's view is a vector, consisting of the most recently observed value (by that agent) for each state variable. However, that update will occur at different times for different agents, due to message latency.

We say that agent \(i\) controls some subset of the state variables, \(S_i\), when agent \(i\) broadcasts values for those state variables to all other agents. In the pursuit problem, each agent controls its own location. The sets \(S_i\) partition the set of all state variables, so each state variable is controlled by exactly one agent. Note that controlling a state variable does not mean determining its "true" value. Each agent's view of the state variables is equally valid. For example, \(p_3\) controls its location, but other agents' views of \(p_3\)'s location could be equally important in determining if \(p_3\) is close enough to capture the prey. Since POMDP and MDP models assume that there is one unique value for each state variable at each point in time, the natural idea that "location of \(p_3\)" should be a state variable no longer applies. Put another way, any single agent's view of \(p_3\)'s location does not capture all of the state information about \(p_3\) that is necessary to determine if it is close enough to capture the prey. Instead, all of the agents' views of \(p_3\)'s location are potentially relevant.

In existing multiplayer games, the exact method for determining game outcomes (e.g., a kill in a FPS or a capture in the pursuit problem) from the differing views of the world is hidden from the agents, buried in the code for the game itself. This is in sharp contrast to POMDP-based models, where the reward function is an important part of the system, utilized by the agents when planning their policies. In real games, there are essentially two reward functions at work. The single-view reward function maps one single view of the state variables to a reward. For example, given a location of the prey and locations of each predator, the single-view reward function determines if the predators have captured the prey. This single-view reward function is known to the agents, and it describes their objectives in the game.

Given that each agent maintains its own view of the game state, it is not possible to design a single-view reward that matches how the (distributed) game determines the outcome. Thus, in real games, there are times when an agent is rewarded or punished in a way that is incompatible with its view of the world and the single-reward function. For example, as a prey, my current view of the world says that the predators are still five meters away, and the single-view reward function indicates that I am still alive, yet the game system registers me as being captured. This is where the second reward function, the game-wide reward function, is noticeable. The game designers embed some mechanism for determining capture events into the game, which could be based on other agents' views in addition to mine. While the game-wide reward function is typically engineered to be as close as possible to the single-view reward function, increased latency will lead to greater differences among the agents' views, which could make it impossible for the game-wide reward function to match the single-view reward function from all of the agents' perspectives.

To see why this distinction between the single-view reward function and the game-wide reward function is often glossed over in networked, multiagent systems (such as games), consider the functions that determine the dynamics of a POMDP-based model in a single-player, non-networked scenario: transition function, observation function, and reward function. We argue that the transition function is the most central component: changing the transition function will necessarily change the system. In the pursuit game, this would be changing the actions that the agents can take. In contrast, one could argue that, given a transition function, if one changed the observation function, the environment would still be the same, but the agent's sensors would have changed. Thus, while changing the observation function will change the agent's policy, it would not change the "world."
Similarly, the reward function can be considered a third layer: given a transition function and observation function, changing the reward will alter the agent's goals and its policy, but the world and the sensors remain the same.

When the game developers engineer a game-wide reward function that closely matches the single-view reward function under conditions when latency is small compared to \( t \), the agents can effectively ignore the distinction and treat the single-view reward function as determining their objectives. Since the transition and observation functions are not different, the system appears as if there were no game-wide reward function. However, as latency increases, we anticipate there is no game-wide reward function that will match all of the agents' single-view reward functions. In such situations, the agents must account for the effects of latency in order to maximize their reward as determined by the game.

In networked games, developers use a variety of methods to account for different states. Generally, developers make one machine, often called the “server,” hold state used for scoring decisions. Of course, the corresponding “client” machines’ state will not necessarily match the server. To minimize the effects of this mismatch, a variety of prediction methods are used so that clients can anticipate the state of other clients and thus the server. Predicts can be, and often are, wrong. Thus, prediction is no guarantee of coherent state among agents. Designers add game elements, such as explosions or gun inaccuracies, to try to hide inconsistencies seen by players.

While the single-view assumption of POMDP-based models is the most significant incompatibility, it is not the only one: these models assume that decision making is performed within a well-defined epoch by all agents. One way to construct this is to have a coordinating signal broadcast to all agents that says, “calculate now.” That signal is an observation and is subject to latency like every other observation. Clearly, this destroys the real-time character of the game, as all agents would need to lengthen \( t \) to match twice the highest latency experienced by any agent.

Thus, we believe that Dec-POMDP is ill suited to networked NPCs because of the assumption of either shared memory or zero latency to make system-wide state consistent or to broadcast an agent coordinating signal. Neither holds in our situation.

**Summary**

In distributed, mixed human-agent systems such as networked games, latency prevents achieving a consistent, shared view of the world across all agents. Neither common-memory algorithms nor POMDP-based models, which assume one true value for each state variable at each time step, fit these systems. We are working on a framework for describing such systems, one that can account for the mutually inconsistent, yet individually correct, views of the different agents. The framework should allow the agents to reason about their actions, including how those actions affect other agents' views of the world, which will depend on the latency in the system.

We believe latency concerns described in this paper extends beyond networked gaming. Any system that contains a collection of (D)AI agents networked with humans in a real-time problem could face similar concerns. For example, autonomous vehicles operating on the same roads as human drivers look similar to a dual of the pursuit problem, where all agents avoid each other. Each agent, human or autonomous vehicle—which is itself a DAI agent—has a limited view of the world that cannot be fully transmitted to all other agents due to latency and decision-making time constraints. Furthermore, decision making cannot be synchronized. Unlike computer games, there are any number of environment variables (e.g., weather, traffic conditions, deer, etc.) that complicate the problem. Algorithms in autonomous cars must account for latency to prevent loss of property or life.

**Acknowledgements**

Andrew Cunningham developed simulations used in portions of this work. Bradley Holbrook and Matthew Keibler contributed to earlier work on state inconsistency. Portions of this work were supported by the II-VI Foundation. Opinions expressed in the paper are not necessarily those of the II-VI Foundation.

**References**


