Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-038 May 27, 2006

# What the Assassin's Guild Taught Me About Distributed Computing

Jacob Beal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Chapter 1

# What the Assassin's Guild Taught Me About Distributed Computing

**Jacob Beal**
MIT Computer Science and Artificial Intelligence Lab
jakebeal@mit.edu

Distributed computing and live-action roleplaying share many of the same fundamental problems, as live-action roleplaying games commonly include simulations carried out by their players.

Games run by the MIT Assassin's Guild are particularly illustrative of distributed computing issues due to their large scope and high complexity.

I discuss three distributed computing issues addressed by Assassin's Guild game design—information hiding, error correction, and liveness/consistency tradeoffs—and the relevance of the solutions used by game writers to current problems in distributed computing.

## 1.1 Introduction

I am a computer science researcher, and one of my hobbies is writing and playing live-action roleplaying games with the MIT Assassin's Guild. Historically, it seemed clear to me that these were very different types of nerdy pursuits. Recently, however, I began to notice connections between problems in game design and distributed computation.

This paper is an attempt to set down those connections clearly and to begin a discussion between the two fields. Game design can clearly benefit from a precise understanding of the problems it engages. Perhaps less obviously, the

study of distributed computation can benefit from understanding the methods employed by game designers.

I will begin by explaining why it is reasonable to think about live action roleplaying games in terms of distributed computing, and how this is particularly the case for games run by the MIT Assassin's Guild. I will then give three examples from game design that illustrate the connection well: the tea impossibility result, recovery of destroyed state in tunnel systems, and liveness/consistency tradeoffs in ranged combat. Finally, I will discuss the lessons for distributed computing that can be drawn from these examples.

## 1.2 Why is LARP Distributed Computing?

In a roleplaying game, each player takes on the persona of a character in the imaginary world of the game. In live-action roleplaying (LARP), the game world is projected into the real world and some types of game action are physically acted out by the players. For example, if a classroom is used to represent a building, then a player moves her character into the building by walking into the classroom.

This is an example of physical embodiment, where the real world directly represents the game world. Other aspects of the game world may not be represented in the real world at all, or may be represented as information managed by a single player, or by the *game masters (GMs)* who are running the game.

Most LARPs also involve simulations whose execution is distributed across many different players. The economy of the game world, for example, may be simulated by players exchanging cards representing quantities of trade goods.

Distributed simulations are not rare. It is difficult for players to maintain suspension of disbelief (the feeling of immersion in the game world) when they have to pause for more than a few seconds in the real world while what happened in the game world is sorted out. As a result, every simulation involving time scales shorter than a few hours tends to be distributed.[1] In a complex game, many simulations will be taking place simultaneously, as the characters fight with guns and swords, command armies, explore ruins, seduce each other, influence public opinion, and so on.

A group of players carrying out a distributed simulation is equivalent to a network of devices executing a distributed algorithm: each player is a device, and two players capable of interacting are connected by a link in the network.

This network is a challenging environment for distributed algorithms:

- Small computational capacity: although players are intelligent, faithful execution of rules is slow, no more than a few operations per second.
- Small working memory: players can remember only a few digits accurately, plus a somewhat larger amount of context-cued rules. Much more can be stored unreliably, or reliably on high-latency storage (e.g. paper).

[1] The exceptions tend to be in games with only a few players and where all of the players are confined to a small space.

- Slow communication: information exchange between players is limited by the rate at which humans act and interpret sensory information.
- Partially synchronous execution: even with watches, it is futile to try to synchronize players tighter than a few seconds.
- Changing network topology: as players move through physical space, their ability to interact with one another changes.
- Poor network connectivity: it is rare for the network of a game to *not* be partitioned if there is any significant amount of space available to players. Cliques of players wishing to act privately will regularly move out of contact with the rest of the game, partitioning the network. In a large game space, a clique working secretly may not come into contact with any other players for hours or even days.
- Byzantine failure: although cheating is virtually non-existent, honest mistakes and confusion about the rules can cause arbitrary failures in communication and computation. These failures are most likely to occur when they matter most because the players are awash in adrenaline during critical moments in the game.
- Differing incentives: games usually involve competition between players, creating groups of players with the incentive to manipulate the simulation to benefit themselves at the expense of others. Although cheating is virtually nonexistent due to social pressure, players will happily abuse one another within "the spirit of the rules."

Most games also encode part of the simulation in inanimate objects, such as a sign or an envelope taped to the wall. Viewed from a distributed computing perspective, these are shared storage devices with no computational capability. These storage devices experience arbitrary stopping failures when destroyed by janitors, disgruntled grad students, etc.

This environment is a fierce stress-test for distributed computing algorithms—almost unrealistically so for many current networking problems. Future computer networks, however, may have much more in common: as computers become smaller, it is more difficult to make them error-free; as they become faster, communication delays become relatively longer; as they become more embedded in our environment, energy concerns limit their processing capability and they become more mobile; as our networks span larger areas, they contain more groups with competing interests.

### 1.2.1 A Brief History of the MIT Assassin's Guild

The MIT Assassin's Guild is of particular interest to our examination of the connection between LARP and distributed computing due to the large scope of its games and its long history of a highly active design and analysis community.

Founded at MIT in 1982, the Assassin's Guild is a student group that runs LARPs. A typical game involves 20 to 60 people—a few extreme examples had under 10 or over 100. The length of a game ranges between 2 hours and 10 days of continuous play—the shortest known game ran for 7.5 minutes[2], and the

longest continuously for a full month[10].

Games often run throughout a large fragment of the MIT campus, covering areas 1 kilometer or more in diameter. MIT's unique architecture of interconnected buildings and tunnels further increases the effective distance between game areas, particularly since many parts of campus used in games have no radio or cell-phone reception.

The Assassin's Guild has a strong tradition of gamewriting and analysis: unlike many LARP groups, most Guild games are only run once, and almost all Guild games are written by Guild members using a set of rules customized for that particular game. As a result, over 300 unique games have been run since the founding of the Guild. The culture of the organization encourages research and development as well: it is traditional to open a game for dissection on the Guild mailing list after it finishes, explicit gaming experiments are often run,[2] and the Guild holds a yearly workshop, "The Ides of March," on the subject of gamewriting.

In the past 10 years or so, this trend has intensified, with a heavy focus on infrastructure to support gamewriting. Although not official, the Standard Rules[9] have gained general acceptance, providing a well-tested base set of game rules which most players are familiar with. On the more technical side, two competing software packages, GameTeX[8] and Template[7] provide suites of tools for development, production, and typesetting of games.

All of these combine to produce a community engaged in research and development of LARP technology. The distributed simulation rules we are interested in are a major area of investigation, due to the difficulty of the issues involved.

## 1.3 The Tea Impossibility Result

The tea impossibility result is what first caused me to realize that LARP writing and distributed computing shared some of the same problems. We were engaged in writing Xiaolong[4], a game set in mythic China, so drinking tea was an important element of the game—for example, every meeting had to have tea served, and no contract could be signed without drinking tea. Since the game involved lots of politics and intrigue, it was possible for people to poison one another, and tea was a natural vehicle for poison and other drugs.

This seemed simple enough. There is a standard method used for dealing with ingestibles in the Guild: the substance is represented by a folded slip of paper, with a visual description on the outside, and the effects of ingestion hidden on the inside. When somebody commits to ingesting the substance, they open the slip, read the effects, and destroy the paper. Thus, only the person who drinks "a clear, odorless liquid" knows whether it was water or LSD.

Sitting down to write the poison rules, however, we came across a curious dilemma. No matter how we arranged them, we couldn't create a distributed simulation satisfying these four constraints:

[2]For example, the *X-Games*[3, 1] and *Guild Camp*[5, 11, 6] series

- Any number of players can secretly add a substance to a teapot.
- Every cup of tea poured from a pot contains the same mixture
- The effects from drinking can be written legibly on a slip of paper that fits under a teacup.
- Nobody can know what a teapot contains without drinking from it.

Frustrated, I proposed that it was provably impossible, and indeed it is. To see why, let's formalize the system slightly. Since every cup of tea has to be identical, the teapot needs to have one or more distinct collections of identical slips, each folded so as to hide one or more sets of information. We can thus represent a teapot as a set of $k$ shared read-modify registers. The registers are initially blank, and modifying the contents involves first reading then replacing the contents (because you can see what's already written on the hidden spot you're writing on, and we're assuming that the modification is carried out by a player).

Drinking a cup of tea is equivalent to an atomic read from all $k$ registers. Adding a substance is equivalent to a modify operation on a register. In order to keep the contents of the teapot secret, modify operations must happen only in blank registers (otherwise one or more substances are revealed without drinking).

The register a player modifies must be independent of the contents of the teapot—otherwise, it would be possible to test whether a teapot has been modified by attempting to add a new substance. Collisions can therefore only be prevented if there is one register per player—hashing isn't a reasonable approximation because game dynamics are strange and there is no good way to predict which players are unlikely to modify the same teapot.

The problem thus hinges on the physical size of the slips of paper implementing each register. Past experience has led the Guild to use slips no smaller than 0.75cm by 8.5cm, folded in half to conceal the contents. With a minumum of 6.375 $cm^2$ of paper per register, ensuring no collisions in a 40 person game (as ours was) would require a minimum of 255 $cm^2$ of paper—nearly half a US standard sheet—folded intricately to allow each register to be accessed independently. That much folded paper does not fit under a teacup.

In the end, we did an ecological analysis of the role of tea and poison in the game to determine which constraint to weaken. We then solved the problem by changing the read-modify register to a read-write register: teapots could be dumped and refilled, but not added to. Of the many possible solutions considered, this was one which provided the best tradeoff between function and cost given the relative importance of poison tea in the game.

Although tea is a special case, similar problems are encountered any time players need to jointly modify hidden information. The solution in each case is different. For example, competitive puzzle solving generally weakens the ignorance constraint, allowing a player to know whether another player has already accessed a register, but not what they have done with it.

## 1.4 Recovery of Destroyed Tunnel State

A *tunnel system* is a common feature of Guild games: a network of difficult to access locations including distant parts of campus, usually filled with useful items and dangerous traps. Tunnels have wildly varying purpose, contents, and rules, and can be instantiated in the game world as anything from secret passages in a castle to a radioactive jungle.

All tunnel systems, however, share the problem of conservative transactions in the face of error. Because the tunnels are far-flung, the items they contain are stored in envelopes taped to the wall which occasionally go missing (thrown out by janitors, etc). When, for example, a player trying to steal the Maltese Falcon finds out that the envelope representing its display case has gone missing, he informs the GMs and has them recreate it. The problem is determining whether the Maltese Falcon was in the envelope when the envelope was destroyed, or whether somebody else already stole it.

More formally, the distribution of items in the game is manipulated through transactions that conserve their number.[3] When an envelope disappears, it is equivalent to the device experiencing an error that zeroes its state. The GMs need to reconstruct the destroyed state using information from the rest of the game.

The usual method is to poll the game to gather the transaction history of the device, which allows the GMs to reconstruct its pre-error contents. The poll must be executed quickly—detection of an error generally implies that players are waiting to interact with the device. At the same time, it must not interfere with the progress of the game or reveal secret information (such as its contents or sometimes even its existence).

Conducting a rapid poll of all the players in the game is often prohibitively expensive. Instead, game designers act preventatively and use a layered defensive design to minimize the likelihood that they will ever have to conduct a rapid poll of the entire network. A typical game employs the following layers of defensive design:

- Long-Term Prevention: the base error rate is reduced by building awareness and goodwill with the community. For example, envelopes say when they will be removed, and careful post-game cleanup helps prevent envelopes from being torn down during future games.
- Short-Term Prevention: the particulars of item and device deployment affect error rates. Error is reduced by deploying envelopes with smaller siszes, away from doors, in areas with stable temperature, etc. Using items with physical representations (as opposed to just a paper description) also reduces error rate.
- Error Tolerance: item economies are generally designed to tolerate some variance in number of items, so correct reconstruction of pre-error state matters only for a small number of key items. Building redundancy into player plots also reduces the impact of errors.

[3] Allowing players to create or destroy items does not affect the problem.

- Monitoring: Players like to tell stories to the GMs during the game, and items with low error tolerance are often important to their characters and will turn up in the stories. A few critical items may explicitly require transactions to be logged with the GMs in advance or following completion.
- Narrowed Scope: the GMs can exploit access controls, group structure, and other information to narrow the set of players polled for a given error correction.

This layered design, while not affecting the worst case cost, lowers the amortized cost and can often prevent an worst cast execution from ever occurring.

## 1.5 Liveness and Consistency in Ranged Combat

Combat between characters is usually the most time-sensitive simulation in a game. A group of players fighting must be able to establish consensus on the sequence of operations carried out, despite the fact that the battle may include many people spread over a large area of space. This is not a simple problem. A combat-intensive game can easily develop a battle involving dozens of people spread through several buildings and lasting for many minutes. Worse yet, the use of semi-automatic and fully-automatic toy weapons means a single participant can easily be involved in the resolution of a dozen or more concurrent operations.

At the same time, resolution of each individual operation must be rapid and wait-free: otherwise a player can end up "stuck" resolving one action while other players act to put him at a disadvantage. The problem is further intensified by the tendency of some players to panic and forget rules during a stressful situation, essentially causing them to freeze up while they re-read the pertinent section of the rules.

This boils down to a classic distributed computing problem of tradeoffs between liveness and consistency, since stress-induced errors mean that we cannot guarantee both liveness and consistency. Impairing liveness leads to "stuck" players being unfairly disadvantaged because of the failure of an opponent. Impairing consistency leads to disputes between combatants about what happened, which may also impair liveness!

Guild designers have settled on a set of rules for ranged combat[4] which guarantees neither liveness nor consistency, but delivers consistently satisfactory results by managing the conditions under which failures can occur.

The rules are simple: attacks are represented by projectiles from a toy gun or harmless thrown objects such as ping-pong balls. The first object the projectile contacts is hit by the attack: if it hits a player, the player hit secretly calculates its effect based on projectile type. Shots that hit long hair or loose clothing don't hit; shots that hits an item the player is carrying hit the player. In case of

[4]Hand-to-hand combat is still a matter of active debate.

question, attacker calls the shot; in case of a serious dispute, all action is halted while things are sorted out.

Considered from the perspective of distributed computing, it is surprising that these rules work at all, since there is barely any attempt to assure liveness or consistency. Yet they do work: disputes of any sort are rare, and disputes that interfere with the game (e.g. by forcing a halt) are essentially non-existent.

To understand why this is, we must carefully examine how inconsistency leads to the adverse consequence of disputes between players. We will start with the empirical observation that disputes over combat arise when *a player believes the outcome of combat is affected by an opponent's unfair advantage.*

First, notice that if the target takes a hit which the attacker believes was a miss, that is an unfair disadvantage and will not lead to a dispute. Second, notice that a dispute arises not on the basis of fact, but on the basis of belief, so only inconsistencies detectable by the players can lead to disputes. Finally, notice that disputes are about outcomes, not individual projectiles: individual projectiles only matter because of their cumulative effect in causing a player's defeat.

Now we can start to understand why the ranged combat rules work. The embodiment of attacks as physical projectiles restricts the consensus problem is to the question of where each projectile first hits, and we can consider only two player combats without loss of generality. Most shots are clean misses or clean hits, and the rules about hair, loose clothing, and items means that the visual perception of the attacker is likely to match the tactile perception of the target. Moreover, the attacker can often see the target notice a shot impacting. Thus, disagreement is already infrequent.

There are three ways that inconsistency can occur: the attacker thinks a projectile missed and the target thinks it hit, the attacker thinks it hit and the target thinks it missed, or both think it hit but the target miscalculates the effect. In the first case, the target is placed at an unfair *disadvantage*, and so no dispute will arise. The other cases are dealt with by the attacker calling the shot: because the attacker calls the shot, a few quick phrases can establish the belief that the attacker and target agree on whether the shot hit. Usually this leads to consistency, but errors in communication and calculation can still lead to inconsistency. For this to lead to a dispute the inconsistency must be detectable by the attacker, but since it is only observed as a cumulative effect in the defeat of the target, and since the attacker generally has some uncertainty about the target's resources, a small rate of inconsistencies is undetectable.

Without disputes, the threats to liveness are never realized, and thus the Guild's ranged combat produces results equivalent to those of a consensus mechanism which guarantees liveness and consistency.

## 1.6 Conclusions

As we have seen, the rules of live action roleplaying games can be analyzed as distributed computing algorithms. Surprisingly, however, the MIT Assassin's

Guild rules solve problems in ways that will feel rather foreign to most students of distributed computing. Rather than provide guarantees, a Guild designer working on an algorithm is likely to look for ways to make exception cases rare and decrease their impact on the larger system. It is an open question whether this sort of approach can be harnessed to help with problems in distributed computing. Clearly, however, there is a relationship between the two fields which invites further investigation.

# Bibliography

[1] Bates, Danny, Philip Tan, Eddy Karat, and William Lowenthal, *X-Games III*, MIT Assassin's Guild (2005).

[2] Beal, Jake, "Partitioning germany: A dadaist interlude", *Guild Camp*, (J. Beal and B. Noorani eds.). MIT Assassin's Guild (2002).

[3] Beal, Jake, Jennifer Chung, and Geoff Schmidt, *X-Games*, MIT Assassin's Guild (1999).

[4] Beal, Jake, Joe Foley, liz smith, and Jade Wang, *Xiaolong*, MIT Assassin's Guild (2006).

[5] Beal, Jake, and Benazeer Noorani eds., *Guild Camp*, MIT Assassin's Guild (2002).

[6] Boylan, Laura, and Beth Baniszewski eds., *Guild Camp III: British Invasion*, MIT Assassin's Guild (2004).

[7] Brown, Jeremy, and Jamie Morris, "Template", Software Package (2001), Version 1.5d, `http://web.mit.edu/jemorris/Template/v1.5d/`.

[8] Clary, Ken, "Gametex", Software Package, 2001 version. `http://web.mit.edu/jakebeal/Public/GameTeX/`.

[9] Morris, Jamie ed., *Standard Rules for MIT Assassin's Guild games (2004-2005 edition)*, MIT Assassin's Guild (2004).

[10] Soderstrom, Melanie, *Midsummer Night's Gathering*, MIT Assassin's Guild (1998).

[11] Tan, Philip ed., *Guild Camp II: This Time It's Not Guild Camp I*, MIT Assassin's Guild (2004).