Distributed Tic Tac Toe: Peer-to-Peer Team-Based Gaming System

Panagiotis Antoniou, Emily Band, Paulius Dilkas, Dafin Kozarev, Joshua Styles

***Abstract*—This article describes a Distributed Multiplayer Tic Tac Toe software game written in Java using the RMI API. While previous implementations attempting to serve a similar purpose exist, none of them boast good design or efficiency and neither of them deliver the same functionality. The developed piece of software allows clients to play locally or globally a game of the classic tic tac toe - but with a twist. Players are separated into teams and decide on what the next move should be by voting. The implementation is derived from a design that follows the distributed applications guidelines and is prepared to handle corner cases such as client disconnects for example.**

I. Introduction

Coding is the main occupation of many people nowadays, and it is a past-time for many too. Most beginners get their first steps in writing code following the local single-threaded model. Often, it can be difficult for new programmers, or even those with more experience, to abstract the key knowledge they have obtained using this model and then apply it in a different way.

Distributed software requires a new look on the system the number of devices has gone up and with it so have the number of processes, concerns and complexities. Some concepts have now changed and need to be dealt with differently; One such example if this would be mutual exclusion. We believe programs that serve as examples and focus on simple local tasks turned into distributed ones can help many people trying to make this step with a hands-on experience of what they are preparing for. Seeing theory in practice is a proven technique and there is evidence for this in every university course that features practical work. This is especially true where games are concerned, proving to be an effective means of communicating programming methods and techniques in an engaging way [**?**]. For this reason, the team has decided to take a simple, well known game and develop it into a distributed system; Tic Tac Toe played as a peer-to-peer, team-based gaming system where the players are organised into teams and required to vote on moves for their team.

By refraining from introducing additional complexity as part of the goal of the project, and allowing the users to see the distributed nature of the software being demonstrated via a simple game, the focus is shifted towards the implications of the distributive aspects of the system and the way they are handled. The finished product can serve to entertain the end users, as well as increase their curiosity and insight about computers and distributed systems. Using Tic Tac Toe as a foundation to demonstrate a distributed system benefits from both the simplicity and familiarity of the game itself, meaning no complex rulesets or in-depth mechanisms need be explained to the users, allowing the distributed nature to be a novel way of experiencing a widely known game.

II. System Design

Tic Tac Toe is a simple game played on a grid of 3x3, consisting of two players. One player uses an X marker and the other player uses an O marker to select a grid space to occupy. The objective of the game is to take turns until one player manages to occupy three grid spaces in a straight line, be it horizontal, vertical or diagonal.

The proposed design takes this simple concept and uses it to demonstrate a distributed system, whereby the two players are instead two teams of up to 5 players, which take turns to vote on where their team should place their markers. Each team has a designated ”leader” which communicates data with their team’s peers, and the opposing team’s ”leader” to synchronise current gameplay state.

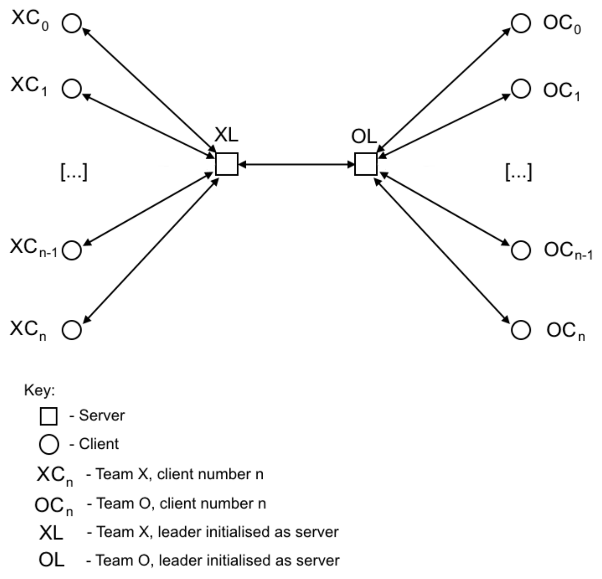
XC$_0$
XC$_1$
[...]
XC$_{n-1}$
XC$_n$
XL
OL
OC$_0$
OC$_1$
[...]
OC$_{n-1}$
OC$_n$
Key:
□ - Server
○ - Client
XC$_n$ - Team X, client number n
OC$_n$ - Team O, client number n
XL - Team X, leader initialised as server
OL - Team O, leader initialised as server

Fig. 1. Topology for Distributed Tic Tac Toe

The leaders communicate the vote of their team to the opposing leader so that they may pass that result on to the peers of that team, as pictured in figure 1.

By considering the fact that only the two leaders communicate about the votes between the two teams, then the system can easily break if one of the leaders crashes.

In order to avoid that and provide a more robust system we chose to implemet the Bully Algorithm on each of the players so they will be able to detect a leader crash if they do not receive a vote reply from them in a certain period of time.

After that, they can easily call for an election on older processes than them and wait for a leader to be announced on a different period of time.

If no leader is announced on that period of time or there are no older processes, then that process can declare itself leader to the rest of the team and the opponent leader.

III. Implementation

The implementation follows the RMI distributed object model. A GameInt interface extends java.rmi.Remote and lists remote classes.

A Game class implements GameInt and extends the UnicastRemoteObject. It is an analogy for what a player knows and does during a game. It provides an implementation for all the listed classes in the interface which include the functionality of requesting the player to make a play, setting the leader of the game, several methods to interact with the Role of the player and a main method that connects to other game instances. C urrently, when running the game, the user passes it a local instance name and the name of another instance as arguments. Each instance then uses this information to do an RMI Registry lookup and get a reference to the other client. The Role interface allows the application to differentiate between Leader and Player classes which extend it.

It utilises the State design pattern to allow players to seamlessly have their role changed for example when the current leader disconnects.

A Leader is responsible of collecting the votes from all of the members in their team in the turnStarts() method. This is done asynchronously using a separate thread for each member an instance of VoteThread. Each thread collects a vote from one assigned player then writes it into a corresponding position in a shared AtomicIntegerArray that they have all been passed by the Leader. This ensures the main Leader thread can then read these values.

Each Leader then informs all players in the game about the vote of his team. A Player instance represents a regular player that only must vote.

Each regular player also has a mechanism to detect if a leader has not contacted them to choose a play for too long. If a time-out is triggered, a player calls for a new leader election. To avoid many simultaneous election calls, each regular player waits 5 minutes plus a randomised amount of time.

All plays are recorded on a GameState object instance that acts as a game board. After each turn it checks if the game is over and relays this message to the leaders which can then inform the members of their team.

IV. Performance Evaluation

V. Architecture Evaluation

While Java RMI provides a simplified approach to object manipulation over distributed systems, it does force the use of TCP sockets by default. The reliable, ordered delivery of data provided by TCP is desirable for distributed applications, however, this choice of transport protocol does have significant implications for application topology.

The ongoing shortage of IPv4 addresses has led to a large proportion of machines on the Internet to be obscured by Network Address Translation (NAT) middleboxes. Machines hidden behind one of these middleboxes cannot be contacted directly without the use of techniques like hole punching, which is a significant problem for establishing an entirely decentralised peer-to-peer system.

Hole punching for NAT traversal requires the use of an external, publicly-accessible proxy server to distribute the public-facing IP address and port combinations (addr:port) of NAT-obscured clients which want to establish a peer-to-peer session. This traversal method is used by many popular distributed applications such as multiplayer gaming, Skype, and peer-to-peer filesharing. This technique takes advantage of the behaviour of NAT middleboxes to create and maintain an 'open' addr:port combination - without this, a fully decentralised peer-to-peer session cannot be established.

As mentioned above, Java RMI uses TCP for its operations by default. While it is possible to modify the sockets created by RMI calls to use UDP instead, this requires extensive work using RMISocketFactory which is beyond the time allowed by the project.

UDP is the more commonly used transport protocol for peer-to-peer applications - one socket can be used for both sending and receiving messages from multiple clients without the need for establishing a fixed connection between two endpoints. Although adjustments would need to be made to allow for unordered or lost messages, UDP would allow us to implement a topology with good fault tolerance and Internet-wide access, as displayed in figure (INCLUDE IMAGE HERE).

An alternative approach is to attempt TCP hole punching instead. This requires creating four TCP sockets with bound to the same local addr:port combination (ie. with SO_REUSEADDR enabled) to establish a connection with a peer through a proxy server, as described in section 4 of a study by Ford, et al (INCLUDE CITATION HERE). Not only is this a significant overhead, with four sockets and multiple ports required for each peer a client wants to make contact with, it is also more complex to perform than the UDP equivalent; our simplified Python-based tests could not establish contact between peers using the instructions described in the Ford paper.

Without significant adjustments to the type of sockets used by Java RMI, a better understanding of TCP hole punching, or the universal adoption of IPv6, our application is currently limited to operating within local networks only. This is acceptable for its use as an educational tool, where it can be used within corporate or university networks freely, but the lack of NAT traversal largely excludes its use as a casual multiplayer game given that the vast majority of homes obscure several devices behind an ISP-provided NAT middlebox.

While we are aiming to create a system without unnecessary complexity for educational purposes, NAT is a core component of the modern Internet - all networks programmers will encounter difficulties with it at some stage and having a solution to demonstrate as part of our application would have been very beneficial to people using it for educational purposes.

VI. Related Work

While other multiplayer distributed games already exist, they have not been created with this projects task in mind and therefore do not serve the same purpose. They can be split into two categories very unprofessional and too professional. For example, hobby implementations exist that deal with distributed problems in a bad fashion, promoting bad and incorrect programming practises. Similarly, a single-player-per-team distributed tic-tac-toe game in Java can be found online that does not comply with any design guidelines and puts all functionality in a single class and as such, cannot teach students good practice and has the potential of confusing the learner further about distributive programming [?].

Separately, very in-depth materials that provide state-of-the-art practices can be found [?] but there is a high possibility that they might be incomprehensible to a beginner reader looking to tackle basic distributed concepts to begin with. Furthermore, none of the similar implementations residing on the Internet take advantage of the Java RMI API which must be considered a good starting point for distributed systems learners for it to be taught as part of our degree and is a requirement for this project. Instead, they use different sets of other technologies which can further confuse their reader and introduce a steeper learning curve.

VII. Conclusions and Future Work