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 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. Our implementation allows clients to play a game of Tic-Tac-Toe with a twist: players collectively decide on what the next move should be by voting in teams. The implementation is derived from a design that follows the distributed applications guidelines and is prepared to handle corner cases such as client disconnects.**

I. Introduction

Programming is very much an in-demand profession in the current job market, as well as being a popular hobby. Most beginners take their first steps in writing code with no concurrency. It is often difficult for new programmers to abstract the key knowledge they have obtained to a point where they can independently apply it to a different scenario.

Distributed software engineering principles are evolving quickly — the number of Internet-enabled devices has rapidly increased in recent years, and the level of complexity in creating programs to use these interconnected systems effectively has increased along with it. This complexity arises from core concepts such as co-ordinating resource sharing and fault tolerance — these are challenging concepts to learn for a novice programmer, but they are integral to creating a reliable and effective distributed system.

We believe that programs which serve as examples of turning familiar, relatively simple local applications into distributed ones are key to bridging this gap, providing a hands-on introduction to a field which is becoming ever more important with the deployment of novel concepts like fog computing. The sheer number of programming tutorials available on the internet shows that there is a demand for model solutions for educational purposes. Games in particular have proven to be an effective means of communicating programming methods and techniques in an engaging way [1]. For this reason, we have 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 the moves their team will make.

By refraining from introducing unnecessary complexity into the project, and allowing the users to see an example of distributed 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. 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 3x3 grid, consisting of two players. One player uses an X marker and the other player uses an O marker occupy spaces in the grid. 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 each player is assigned to one of two teams, each of which takes 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.

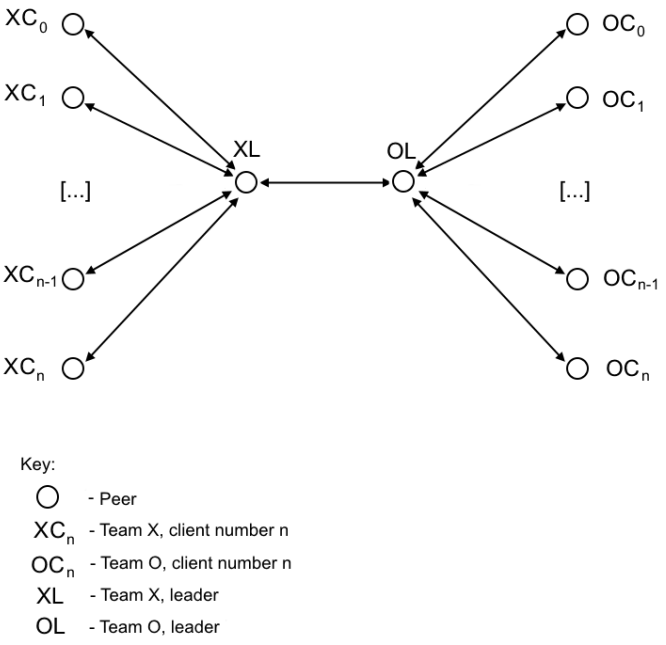

Fig. 1. Topology for Distributed Tic-Tac-Toe

A new game instance tries to join an already existing game specified by the server name and an RMI Registry handle. If a connection succeeds (ie. that machine is paricipating in an active game session), the new player will retrieve the current

game state, initialise differently depending on whether it is assigned to be a leader or not, print the current board state, and prepare for the turn.

The leaders update a common game state at the end of each event before broadcasting this updated game state to all team members. Each player checks if the received play ends the game. If so, it stops all activity and, after the RMI connections close, exits.

III. Implementation

The implementation follows the RMI distributed object model. A `GameInt` interface extends `java.rmi.Remote` and lists all methods that can be called using RMI.

A `Game` class implements `GameInt` and extends `UnicastRemoteObject`. It is the main class for each player, holding most of its knowledge and behaviour. It provides an implementation for all the listed methods in the interface, which include the functionality of requesting a player to make a play, setting the leader of the game, several methods to access role-specific functionality, and a main method that connects to other game instances.

Currently, when running a game, the user supplies a local instance name and the name of another instance as command line arguments. Each instance then uses this information to do an RMI Registry lookup and get a reference to the other peer in an attempt to establish a session. This allows multiple different sessions to run simultaneously on the same network without interference — ideal for educational use, where multiple groups are likely to be working on the same internal network.

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 when required (eg. if the current leader disconnects).

A Leader is responsible for collecting 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 and writes it into a corresponding position in a shared `AtomicIntegerArray`. The leader then uses these votes to decide on a play.

Each `Leader` then informs all players in the game about the most popular vote chosen by their team. In the event of two or more moves receiving an equal number of votes, a leader will randomly select one of them. A `Player` instance represents a regular player that can only vote, it cannot independently generate and broadcast a new game state.

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 several simultaneous election calls, each regular player waits five minutes plus a smaller randomised amount of time.

Elections use the "Bully" algorithm to determine which `Player` instance will become the new `Leader` instance for their team. The other team members and the leader of the opposing team are all informed of the new leader, creating an altered topology to compensate for the failure:

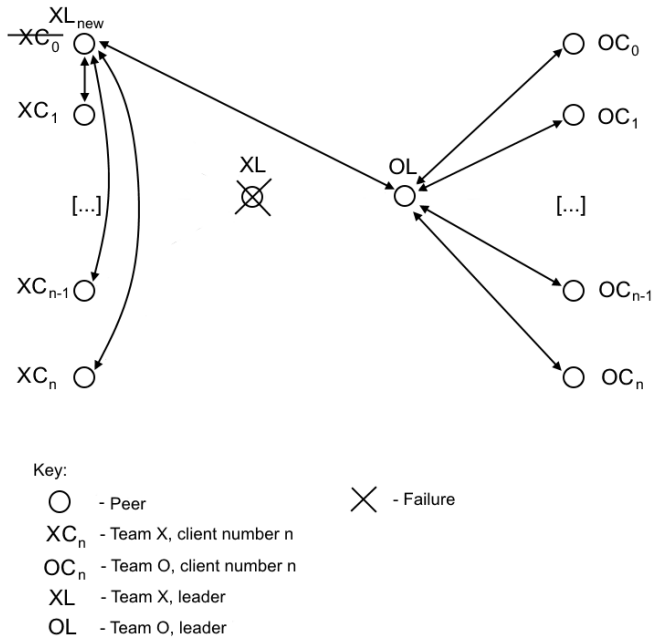

Fig. 2. Topology changes after a leader disconnects

If the failed machine joins the game session again, it will not automatically regain `Leader` status, instead participating as a regular `Player` instance.

All plays are recorded in a `GameState` object instance, which acts as a game board. After each turn, `GameState` checks if the game is over. If it is, the client quietly waits for the RMI connections to time out.

IV. Performance Evaluation

We created a script, `createpeers.py`, to automate the creation of peers for testing system performance. This can create a base instance to start the game, and can then create any number of peers which will attempt to connect to the base instance.

We decided to investigate how the "Bully" election algorithm scales with the number of peers on a team (fig. 3).

The general trend is that the election caller will take a longer time, on average, to process each election vote from each peer on its team. However, the results for 15 and 20 nodes are anomalous. There are several potential causes for this: another process on the machine running these tests blocking one of the voting threads, election threads blocking other threads from accessing a shared resource, and refused connections.

A considerable number of nodes failed during these elections, with this number varying between 4 failures in the 15 node run and 11 in the 30 node run (fig. 4).

These failures were all caused by `java.rmi.ConnectException`, which was thrown in response to refused TCP connections. Java RMI calls require an exclusive TCP connection to a client, which is

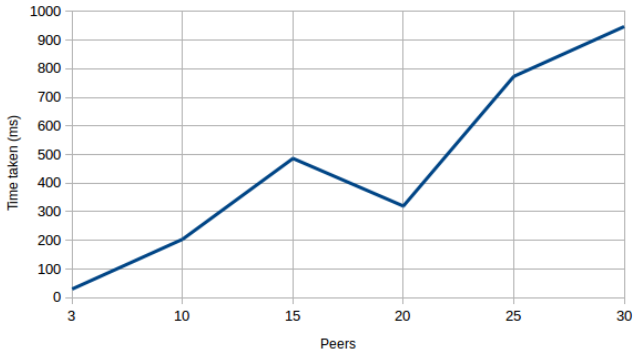

Fig. 3. Average time taken to respond to process a response from each peer in an election call

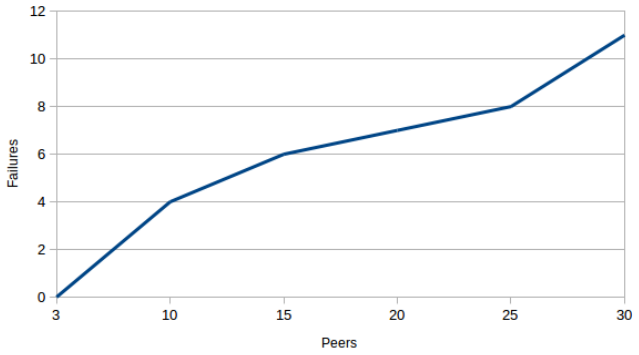

Fig. 4. Number of node failures caused by java.rmi.ConnectException

not suited to the "Bully" algorithm. Although the election algorithm itself is resilient and still elects a new leader from the surviving nodes, these failures strongly suggest that modifying RMI to use UDP sockets or `SO_REUSEADDR`-enabled TCP sockets is required for a more failure-resistant peer-to-peer system. This idea is covered in more detail below in Architecture Evaluation.

The next step would be to repeat these tests in a scenario closer to how this project is likely to be used in an educational setting such as a university lab exercise: several different machines each running between one and three instances of the game. We would record how many failures are caused by `java.rmi.ConnectException` in comparison to running all instances on `localhost` as well as comparing any difference in average time taken to process votes.

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 types of distributed applications such as multiplayer gaming, VoIP, and peer-to-peer file sharing. 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 robust fault tolerance and Internet-wide access:

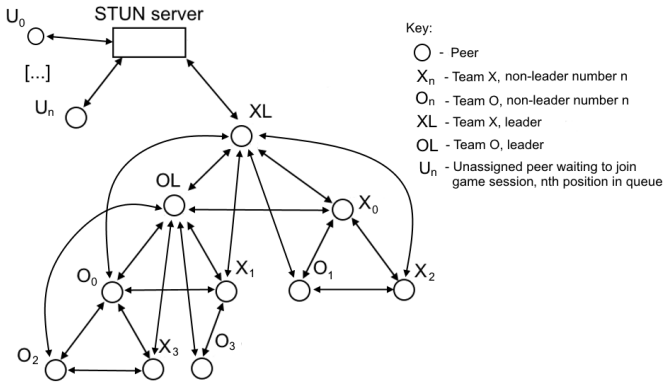

Fig. 5. Proposed Topology for UDP-based Version of System

An alternative approach is to attempt TCP hole punching. This requires creating four TCP sockets 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 paper by Ford, et al [2]. Not only is this a significant overhead, with four sockets required for each peer a client wants to make contact with, it is also more complex to perform than the UDP equivalent - our initial 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 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, 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 extra material for people using it for educational purposes.

VI. Related Work

While other distributed multiplayer games already exist, they do not serve the same purpose as our project. Existing products can be split into two categories: too simple and too complex. Examples of the former generally deal with the difficulties of distributed systems in a bad fashion, promoting questionable design and programming practices. One specific example of this is a YouTube guide to building a single player-per-team distributed tic-tac-toe game in Java [3] — while the material is accessible for inexperienced networks programmers, it puts all functionality in a single class and does not comply with any design guidelines. As such, this resource has the potential to further confuse learners about programming distributed systems when they try to read more complex documentation, which does obey software engineering design principles.

The latter category consists of resources that provide details of state-of-the-art practices and advanced optimisations. An example of this is a version of the game Quake II, modified to run multiplayer sessions more efficiently over a distributed system [4]. This is excellent material for an experienced programmer looking to refine their skills, but daunting to a newcomer due to a high level of complexity in both networking concepts and game logic.

Furthermore, none of the similar implementations residing on the Internet make use of the Java RMI API. Many distributed systems projects use other technologies, such as C or Rust, which have a far steeper learning curve than Java, and are not as commonly taught in universities or used as frequently in enterprise environments.

In addition to being based on a widely-known language, Java RMI has the additional benefit of simplifying processes such as marshalling and socket creation. Our project takes advantage of this to allow people to learn about concepts in distributed systems programming without worrying about added complexities such as objects becoming corrupted due to incorrect formatting, or incorrect use of transport protocols.

VII. Conclusions and Future Work

The produced software serves as a good demonstration of a distributed game. The use of a simple, well-known game turned into a distributed, multiplayer experience highlights the benefits and complexities associated with distributed systems, and allows the technology to be demonstrated in an accessible format, allowing users of all levels to make use of the software without being discouraged by an overwhelming level of complexity.

Distributed Tic-Tac-Toe could benefit from some changes in future iterations, including making the software more robust and granting the users an improved playing experience by addressing certain network issues and gameplay elements.

In the project's current state, leaders are appointed using the "Bully" algorithm when a leader fails. However, there is no support for failure of communication from both leaders, i.e. if both leaders were to crash at the same time. Although the likelihood of this occurring is slim, this possibility still needs to be addressed to improve player experience and to improve the standard of our product as an educational resource.

Another improvement, which we would make in further iterations, would be some form of automatic team balancing — currently, no replacement team members are allocated to an underpopulated team if several peers drop out of a session. Another situation which needs to be addressed is the scenario of all members of a team leaving a game before it is complete, an event which currently has no resolution.

The simplicity of Tic-Tac-Toe, combined with NAT traversal issues, limits our project's appeal as a multiplayer game to people with no interest in programming. However, our commitment to obeying software engineering design principles in Java makes this a valuable educational tool for beginners wanting hands-on experience with engineering a distributed, peer-to-peer application.

References

[1] J. R. Kiniry and D. M. Zimmerman, "Verified gaming," in *Proceedings of the 1st International Workshop on Games and Software Engineering*, ser. GAS '11. New York, NY, USA: ACM, 2011, pp. 17–20. [Online]. Available: http://doi.acm.org/10.1145/1984674.1984681

[2] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," Apr 2005. [Online]. Available: https://dl.acm.org/citation.cfm?id=1247373

[3] M. A. Y. User, "Make a networked tic-tac-toe in java," https://www.youtube.com/watch?v=aIaFFPatJjY, 2015.

[4] A. R. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," in *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings.*, 2006. [Online]. Available: http://www.usenix.org/events/nsdi06/tech/bharambe.html