Automated Poker

Dept. of CIS - Senior Design 2009-2010

Albert Kim
albertmk@wharton.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Barry An
barryqa@wharton.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Jonathan M. Smith
jms@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT
We discuss the design and development of a multiplayer Texas Hold'em Poker game where user-created bots will compete against each other instead of humans. The purpose for this application is to allow users to play more hands than they normally could in traditional poker websites, where each action must be manually performed. This idea is not unlike the shift towards algorithmic stock trading machines in the financial world. We first develop a distributed back-end system, providing the infrastructure for the bots to play at high frequencies on many tables. We then provide tools and an interface for users to implement their own algorithms into unique bots. We create several example bots with various decision-making abilities, which we then analyze collectively for testing purposes and to demonstrate that our application can serve as an alternative to popular online poker sites. In creating this application, we seek to answer several issues surrounding the idea. We first determine whether a realistic poker environment can be mimicked, where participants fairly earn/lose money based on their decision-making abilities. We also determine whether a user can feasibly translate his/her playing strategy into an algorithm. Because bots can play a significantly greater number of hands than a person can, we aim to investigate whether this difference affects the gameplay from a human's perspective. We closely monitor and study the quantitative aspects of the game, such as variance, increase/decrease in profits, and strategy implications.

1. INTRODUCTION
There are currently dozens of popular online poker websites allowing users to play various poker games, including Texas Hold'em, Omaha, and Seven Card Stud. These sites attract tens of thousands of active players from all over the world. There is an ongoing debate (in and out of the playing field) on whether poker is gambling or a complicated game of skill. However, there is no doubt that successful players are able to repeatedly make money from this game through careful probability and risk analysis, in addition to good judgment.

Poker is played with only the money pooled together from its participants. As opposed to other casino games, such as Blackjack, players do not compete against a “house”, but rather, with other players. This aspect of the game is commonly used in favor of poker’s classification as a game of skill, rather than luck. In a game of Hold’em Poker, the dealer will distribute two cards to each player, then sequentially place community cards in the center of the table after each turn of betting. The object of the game is to bet according to the strength of one’s hand against others’ or to fold (give up play without committing any more money) if the odds are not in favor of the player.

Several poker terminologies will be used throughout this proposal. A group of players (usually up to 8 to 10 max) can play at a single table. By sitting at multiple tables (permitted online), players participate in several independent games, each having no effect on the outcome of another. Players will act one by one around a table, and they can bet, call a bet, or fold. Betting is when a player places money in a pot, usually indicative of possessing a strong hand (they could also be faking, or “bluffing”, their strength). When a bet occurs, any following bets or calls for that round are required to at least match amount originally wagered. Players can also fold a hand, essentially giving up play for that round and not having to risk any additional money.

It is very common for experienced poker players to play on multiple online tables at a time (some up to 20 or more) since this reduces variability and provides an outlet for greater returns. In a sense, this act can be compared to portfolio managers or stock traders taking positions in a diversified portfolio to reduce return volatility. In the past decade, an increasing number of investment banks, mutual/hedge funds, and trading firms have been buying and selling stocks through electronic exchanges [6]. One source claims that 73% of daily equity trades is done automatically by machines with intelligent algorithms [11].

These machines have significant advantages over traditional (human) traders. Aside from executing trades in a matter of milliseconds, they require much less manpower and attention to fine details. Algorithmic traders specify algorithms that satisfy their overarching goal and let them run throughout the trading period. By engaging in a huge volume of trades with incremental returns, institutions are able to reap substantial profits. Even during the midst of the recent financial crisis, firms such as Goldman Sachs made record profits due to these electronic trades [4].

Our project seeks to mimic the automated nature of algorithmic trading in multiplayer poker software (particularly Hold’em). Automated decision-making bots are currently discouraged by (and usually banned from) online poker sites, as they may provide an unfair advantage over live players. These programs/bots work in conjunction with statistical analysis software to decide on the best course of actions versus certain types of players. For example, the program might decide that a certain opponent has given up a pot 75% of the time in a specific scenario and automatically try to exploit
that behavior again.

We believe that the future of poker lies in the automated world. Instead of humans playing against one another, poker can be played with user-created algorithms, or “bots,” that compete on the poker table. With rapid advances in artificial intelligence, bots can be designed to play poker in a very similar manner of its creator without human errors or faults (for example, judgment can be clouded by emotion). We want to develop an application that allows users to play fairly against each other by allowing them to create competitive algorithms that can win more hands for them and eventually make money.

2. RELATED WORK

Not much academic research has been performed specifically regarding the online poker world and much less regarding an automated poker offering to the public. However, because of our project’s unique position at the intersection of online poker, high-speed information storage and transfer, and automated decision-making processes, there has been plenty of related research in those fields that we can use to advance our project. Specifically, we explore the research performed on the system technology we want to use for our back-end as well as the bot decision-making abilities and behaviors we want to make available for users playing on our software.

We envision our poker exchange to act very similarly to a high-speed online stock exchange: both take in large amounts of orders (raise, call, or fold for the poker exchange versus buy or sell for the stock exchange), execute these orders, and transfer and store all relevant data in a timely fashion. Hendrshott details the importance of handling these actions via limit order books and electronic communication networks in a trading environment [8]. Our poker exchange will contain analogous systems that will receive/respond to bot requests as well as store and process information during a game.

We considered various parallel computing and distributed caching technologies to implement this, as we felt they would best handle and process the data in real-time. Parallel computing would be useful in our system because most of the calculations will be repetitive, and hence, can be partitioned into table groups on several machines. Distributed caching allows us to store data solely in memory, allowing faster access. Much of the data we will be dealing with only requires temporary storage, and combined with parallel computing, we can distribute only the necessary data into each of the partitions to be worked on.

Rodriguez et al. describes two caching architectures, hierarchical and distributed caching, and compares them in terms of various performance metrics. These metrics include the client’s latency, bandwidth usage, and disk space usage [10].

We also studied several articles that examined the effectiveness of distributed computing over a single large machine. In their article, Ahuja et al. provides a comprehensive explanation of JavaSpaces, a distributed programming tool designed for Java [1]. They describe JavaSpaces as a simple framework that can be used to solve a large class of distributed problems. The key feature of JavaSpaces is that it allows processes to work in parallel with each other and not have to worry about the actions of the other processes. Within JavaSpaces, processes are loosely related, and communicate through a “persistent object store” called a “space” rather than through direct communication. Ahuja et al. then introduces GigaSpaces as “an industrial-strength JavaSpaces implementation,” providing many wrapper libraries that optimizes the application and simplifies development.

Ahuja et al. also introduces a series of experiments that were conducted to test the relative effectiveness between GigaSpaces and ORBacus, a product based on the Common Object Request Broker Architecture (CORBA) standard. A distributed, “parallel insertion sort” application was executed on both platforms, and response times were recorded. The article was able to conclude that GigaSpaces was able to consistently outperform ORBacus. Therefore, Ahuja et al. claims that the JavaSpaces platform was ideal for developing distributed parallel algorithms, which we chose to pursue.

Hancke et al. also details the performance of different distributed systems [7]. The authors calculated the overhead of distributed programming tools, such as JavaSpaces, using a statistical approach. In doing so, their primary motivation is to determine whether such distributed programming is ideal for solving high performance calculations. Their article concludes that although the overhead for JavaSpaces is not too large, there is significant variability, especially when outliers are considered.

These factors will be of important consideration in building our poker exchange, and we hope that our poker exchange implements an effective caching system that reduces many of the problems associated with storing and processing large amounts of data quickly.

In addition to learning about the system component of our application, we must also be aware of the nature of the algorithms that will take place on the platform.

Perhaps one of the first poker-playing bots publicized on the internet was the Poki system, created by Billings et al. [3]. Poki is a poker-playing algorithm designed for limit Hold’em games. Billings describes the significant attributes built into Poki as hand-strength assessment, evaluation of hand potential, bluffing, unpredictability, and opponent modeling.

Put simply, the authors describe a central evaluation system takes in information about the hand, the model it has developed thus far on its opponents, the state of the board, its pot-odds, and various other factors and develops a set of possible actions (for example, 25% fold, 50% call, 25% raise). Then an action selector would select one of these actions at random. Although the Poki system presented a very solid foundation on the necessities of a poker-playing algorithm, it focused on limit Hold’em, a form of poker more constrained and arguably less interesting than no-limit Hold’em. In 2007, Beattie et al. [2] introduced a model designed to play the more popular no-limit variant of Texas Hold’em. This poker bot focuses on three key considerations: hand value, risk, and aggressiveness.

For each consideration, Beattie’s algorithm uses mathematical modeling to decide on the best course of action. For example, Beattie describes risk as a function of the bet size, the pot size, and the size of the blinds. Beattie implemented many of these poker-playing agents and concluded that they performed competently versus “average” human players.

We draw from these works that our system must allow bots to store and update information while playing the game in order to make complex decisions. Moreover, our applica-
tion must provide an API that allows users access to data specific to a certain table. We must find an optimal way for users to access and process these data in their algorithms without infringing upon overall system performance.

Our goal is to combine these prior establishments, both architectural and algorithm-related, as components to build an innovation and efficient poker-exchange system.

3. SYSTEM MODEL

Our primary goal is to create a multiplayer poker application that can conceptually serve as a feasible alternative to popular online poker websites. Because poker is a game played relative to other players’ actions, a player must wait for all of the other players to complete their actions before making his/her action. In the event that the player folds a hand, he/she must then wait until the next set of cards are dealt. Even when every step of this is done automatically, there is inevitably some waiting time for other players to “think”, or to wait until the current hand is over.

This project seeks to reduce that waiting time for a participant to make a decision. We achieve this by automating the decision-making process via bots. We attempt to create a setting in which an individual “algorithm” can be exploited to work constantly by replicating the same table multiple times. This can be compared to taking a position in a financial options or futures contract, where a single contract consists of hundreds of shares. Any small benefit/loss received from the table or option is multiplied by the number of tables or shares, respectively. This allows the bot to play many more hands in a reduced variance setting against the exact same opponents. The tables can be replicated enough times for most bots on average to approach a scenario where the bottleneck in performance is due to thinking time, or minimizing idle time for each bot.

Our poker software/game consists of both the back-end server and the interface (mainly API calls) for users to create and monitor bots. As mentioned before, the server is developed on a distributed platform, taking advantage of the parallelization of data storage and processing. An interface allows users to program their own algorithms via API calls through which the back-end machines will respond to. In addition, it allows the bot creators to monitor the performance and statistics regarding his/her bot and the tables available for play.

Most of the back-end system workflow occurs in-memory across several different machines. A remote application, or the user interface to this distributed application, chooses to upload a specific type of bot to join a poker table against other players. A Grid Service Manager (GSM), which abstracts the partitioning schema from the point of view of the remote application, takes this request and instantiates a new bot, written to the space, containing information regarding the specific bot. When the Table object communicates with the new bot, the bot is notified of this start of a new hand and takes these notifications

from the new bot.

During a hand, each player, or bot, is given two random cards from the deck. This occurs when the table manager process writes a notification object into the space, one for each bot participating in the hand. This notification object contains information about the cards distributed to the players. Another manager process that handles bot actions is notified of this start of a new hand and takes these notification objects and incorporates the information to the bot objects.

The table manager proceeds to immediately write another notification object in the space, informing a bot of its turn to make an action (Figure 2). This notification object contains information about the history of the current hand (there is no information at the start of the hand, aside from the initial pot size from blinds and antes). Again, the bot manager takes this notification object out of the space and feeds information to the respective bot.

It then invokes the bot to make a decision object to write back in the space, containing information regarding the specific action and bet amount if any (Figure 3). The table manager then takes this decision object and processes the new information on the table object. This process repeats until the hand is over.

This workflow all occurs within the context of a single machine. Each machine operates a chunk of the total “space” of the distributed system. Bots and poker tables that are written to the space can be partitioned based on a specified field to ensure it is distributed to the correct machine. A re-
Figure 2: 4. On a certain Bot’s turn, a BotTurnObject is written to the space containing information about the current round. 5. This write event notifies the BotService. 6. The BotService invokes a function in Bot to make a decision based on the new information.

Remote application monitors and controls the objects that are introduced into the space. To this application, the partitioning logic is transparent, and it is simply communicating to a proxy of the space that can automatically redirect requests to the correct partition. Hence, the interface thinks it is communicating with a single, but large, memory instance.

4. SYSTEM IMPLEMENTATION

To implement the above design and system workflow, a third-party product called GigaSpaces is used. GigaSpaces is a platform that allows one to easily distribute a large application across several machines in parallel. It essentially creates a large “cloud” of memory, called the “space” or “GigaSpace”, that is spread across these machines, and allows different services to interact with each other via the space. Services in our project refer to the applications that actually handle business logic and the reading and writing of data objects.

It is important to note that each partition of the cloud operate independently of each other, as the system relies on parallel computing. The cluster of partitions is referred collectively as the “space” since to the outside user or remote application, the partitioning schema is transparent. Each partition has a replicated set of services, equivalent across all machines, and they operate solely on the set of data objects local to their machine. For example, when a user wants to write an object into the space, GigaSpaces will automatically route that request into the appropriate machine.

GigaSpaces is built entirely on Java, which provides some benefits. Likewise, the poker system is built using Java because of the extensive packages available with this language. Since many of the required functions and data structures are readily available for use, development will be relatively easier and faster compared to low-level programming languages. In addition, GigaSpaces builds on top of JavaSpaces and Jini, both built by Sun Microsystems as part of the Java

Figure 3: 7. After invoking Bot’s function, BotService writes a DecisionObject to the space. 8. This write event notifies the TableService. 9. TableService takes the DecisionObject and modifies the data in Table to reflect the most recent action.

Figure 4: Various GigaSpaces Topologies: partitioned, partitioned with backups, replicated [5]
package.

JavaSpaces and Jini allow coordination and distribution of objects across a distributed system set up using the Jini network protocol. GigaSpaces builds an additional layer on top of these technologies to allow creation of a service-based distributed application and data grid, and it abstracts away much of the lower-level networking and distribution aspects.

Because the goal of this system is to play many hands in a given period of time, the design and implementation decisions were geared towards maximizing performance. However, there is also considerable effort to keep this application as modular as possible.

We classify our application into three types of classes: services, data objects, and data access objects. There are two services in the system: TableService and BotService. The purpose of these classes is to handle all of the business logic behind the game of poker. For example, TableService compares two players’ hands and determine the winner. It then distributes the pot (collective bet amount for all players that round) to the winner(s). In our partitioned application, each partition will have a copy of the services running in memory.

Data objects are essentially Java POJOs (Plain Old Java Objects). As their name implies, they primarily serve to store data and hold information about the current state.

Though some of the fields in the data objects are Java Collections, complex fields are otherwise avoided, such as a reference to other objects in the POJOs. Instead, references to their primary keys are used (i.e., BotID). This is a design choice made on the fact that a requirement of the system is to achieve optimal performance. GigaSpaces provides such optimization when querying objects using a primary key or indexed field in the space, much like a database. In fact, the space can essentially be viewed as a distributed database running in memory, where it will be populated by the data objects that store data.

Data access objects (DAOs) were implemented to communicate between the services and data objects. The DAOs contain an instance of a “GigaSpace” variable, which is the proxy to the space. Whenever the services process some logic and need to read or modify objects in the space, this is done by calling the data access objects.

Several “helper” classes that did not fall under the above categories were implemented as well. In particular, the PokerAnalyzer class processes most of the logic specific to the rules of poker, such as determining the winner(s), given a set of players still playing at the end of a hand.

PokerAnalyzer is a library of many helper methods designed to aid TableService in the determination of winners. In addition to being accurate, this library is designed to be highly flexible and efficient, taking advantage of code reuse and paying special attention to the use of effective algorithms.

TableService is a service class that helps process a round of Texas Hold’em from beginning to end. This service has several main responsibilities. First, it is TableService’s job to process all the DecisionObjects written into the space from the BotService and Bots. TableService accepts a DecisionObject and immediately updates the poke...
currency issues and many possible race conditions while maintaining high performance. Our in-memory components only select the table and bot objects that are ready to be processed. In the notification-based operations, if multiple notification signals are produced simultaneously, such as a bot telling a service to process its decision, our program will automatically queue the work requests in the order received and process them based on the number of available running threads. In the polling-based operations, our applications will poll and operate only on the objects that are ready to be processed; otherwise, they will immediately be released.

Our application is also designed to minimize any complications from multiple threads running. Since most of our entire software operates on an in-memory cache, where objects are quickly being read and modified, there may be certain race conditions that could arise. We have ensured that these do not happen by sacrificing a tiny portion of performance by making sure read/write locks are properly secured in even the most remote corner cases. Whereas we could have dealt with these issues by simply restarting a hand when such a problem occurs, ensuring correctness was more easily implemented into our system. When an object is polled, it is completely removed from the space; no other threads will know of its existence thereafter.

5. RESULTS

The preliminary tests of our automated poker platform were done with several test bots, prototype versions of bots that we expect users to submit to our system. With these bots, we were able to perform test runs to determine both the accuracy and efficiency of our platform. Overall, we performed several batches of five-minute test sessions. For each session, we tested a different table setting (9, 6, or 2 player size) and different combinations of test bots.

Figure 5: Sample flowchart for RandomBot showing its decision process. In this case, it follows a 20/30/50% distribution for folding, calling, or betting. For each instance, its actual action will be determined randomly.

Three of the test bots we developed are called RandomBot, SmarterBot, and ModelBot, and their names roughly reflect their abilities. RandomBot (Figure 5) is the most primitive of the bots we created and serves simply as a benchmark for other bots. It makes each of its decisions on a random basis. Each time it needs to make a decision, RandomBot determines its action based on a fixed probability distribution. For example, during the preflop round of betting, it may decide to fold 50% of the time, call 30% of the time, or raise 20% of the time (or otherwise determined by its creator). RandomBot also acts independently of the strength of its own card holdings; it essentially ignores what cards it currently has. Thus, as designed, its decision basis is strictly random and we expect it to perform rather poorly against more intelligent bots.

Figure 6: Sample flowchart for SmarterBot. SmarterBot first determines the strength of its hand, then evaluates the actions taken leading up to its turn, and finally makes a decision.

SmarterBot (Figure 6) is a more detailed test bot with the ability evaluate its own card holdings. Preflop, it evaluates its cards and determines their relative strength against two randomly dealt cards. This measure of strength (and any other future mention of “strength”) refers to the percentages of winning the pot if the two hands are played until the river card. The relative strengths of each hand have already been well-established in the poker community, and we used this information in a simple lookup fashion to determine a certain hand’s strength. If SmarterBot determines that it has a strong hand (for example, lies within the top 20 percentile), then it will proceed to play the hand as long as there hasn’t been an abnormally large bet. Otherwise, it will fold to the bet. Similarly, after the flop, SmarterBot determines its action strictly based on the strength of its hand. If it determines that it has made a high-card pair or better, it will continue to play the hand by either calling the bet or raising initial bet amount. Otherwise, it will get out of the hand by folding. Once again, we reiterate that there is nothing special about playing the top 20% of hands or continuing with a top pair or better; these are simply user-set parameters that can be altered at any time. In this sense, our automated poker platform allows the user to freely model their bots after their own games.
Figure 7: Sample flowchart for ModelBot. After evaluating the strength of its own hand, it then uses its model to analyze the behavior characteristics of each of its opponents, and makes a decision based on both factors.

Our third example bot, called ModelBot (Figure 7), is an improvement on SmartBot and with a more complex algorithm with some basic artificial intelligence capabilities. As its name suggests, ModelBot contains data structures and statistics logic that is able to record and predict the behaviors of its opponents. For example, it keeps track of how often each opponent raises during a certain betting stage, how often they fold to a raise, how often they bluff, etc. After every hand it observes, it updates the relevant statistics for each opponent that was involved in the previous hand. Then, in future hands, it uses its model on each opponent as a basis for its decision. For instance, if it records that a particular opponent has bluffed 0 out of 20 past times on the river card (meaning the opponent has always bet with a strong holding), it will not call a river bet from that opponent without having a very strong hand itself. If, on the contrary, its statistics on the opponent suggests that the opponent bluffs on the river with a very high frequency (say, above 50%), then it will call its opponent’s bet with a much wider range of hands, perhaps a single pair or even a high card. We envision that eventually, most of the winning bots on our automated poker platform will incorporate a model of some kind, as most winning poker players do in real life.

After running our test sessions, we gathered preliminary data on the win rates of each bot when placed in various tables. As expected, ModelBot performed the best, SmarterBot performed second-best, and RandomBot came in last. When we simulated a single table of nine bots (three of each type) randomly positioned around the table, both Model-Bot and SmarterBot consistently came out as winners, while RandomBot was an overall loser. On average, ModelBot achieved a win rate of about 14-16 big blinds per 100 hands, SmarterBot achieved a win rate of about 8-10 big blinds per 100 hands, and RandomBot achieved a loss rate of about 23-25 big blinds per 100 hands. When simulating tables consisting of only two types of bots, we achieved similar results. Both ModelBot and SmarterBot achieved relatively large win rates against RandomBot, and though it varied widely, ModelBot was able to achieve a very small win rate against SmarterBot.

These win rates are not surprising. When all three test bots played versus one another, ModelBot and SmarterBot were much more selective in the hands they played, as dictated by their algorithms. RandomBot, on the other hand, played a much larger proportion of hands without worrying about the strength of its holding. Thus, in a typical hand, there would be one or two ModelBots or SmarterBots with very strong holdings against two or three RandomBots with completely random holdings. The ModelBots and SmarterBots would win those hands more often than not.

The relatively small win rate of ModelBot over SmarterBot is slightly more non-obvious, but nonetheless can be explained. In essence, ModelBot has all the hand strength-determining features that SmarterBot has, but it is also able to keep track of statistics on its opposing bots. However, because SmarterBot is programmed to play solid “ABC” poker and rarely make unprofitable decisions, ModelBot’s model of SmarterBot is simply that of a typical, solid bot, and thus it is rarely able to exploit any weaknesses. Thus, it only achieved an average win rate of around 3 big blinds per 100 hands over SmarterBot.

The users of our automated poker platform are required to submit a bot that can interact “correctly” with our platform. That is, the user bot simply needs to be able to submit a decision object notifying our platform of its action. Whatever algorithms or techniques the bot undergoes to come up with the decision is entirely up to the user. However, we envision that through natural selection, only bots with solid, winning algorithms will continue to play on our platform. Because of the large number of tables bots can play on and high speed of each hand, bots can experience an extremely large number of hands per hour. Thus, any bots with exploitable errors will lose money very rapidly, and only the bots with the most solid and clever algorithms will be able to continue playing. We predict that the eventual majority of user bots will have algorithms consisting of three key elements: an aptitude to determine the strength of its holdings relative to possible opponent holdings, the capability to model its opponents and keep track of its opponents’ decision patterns, and the ability to randomize its decisions to a certain extent so that even the smartest of opponents will not be able to predict its holdings or next moves with complete certainty.

A benefit of our automated poker platform is that it provides a means for users to test their novel strategies before implementing them in their own games, whether traditional live/online or in a future automated platform. Our system provides a fast and effective way to test a strategy for a large number of repetitions. Users will first need to build the important aspects of the strategy into their test bots, and then submit this bot to our platform, which allows it to play versus a variety of other bots with different styles. Then, users can analyze how their bots fared against each type of oppo-
ment, and subsequently determine if their strategy is viable. For example, if one thought that going all in pre-flop with the top 50 percent of hands is a viable strategy in an environment where all stack sizes are less than ten times the big blind, he or she could easily program such a bot and have it play on the “low stack” tables of our platform. Then, after one hour, or roughly 20,000 hands, the user can observe how the bot did in terms of profit and loss, and analyze the bot’s strengths and weaknesses versus different types of bots. He or she can finally develop potential improvements to this bot and ultimately conclude whether this strategy is a reasonable one or not. By using our system, users can gain a better understanding of the effectiveness of any strategy in a very short period of time.

6. SYSTEM PERFORMANCE

In the ideal setting, our automated poker software would have been tested through a large mixture of bots ranging in complexity from making purely random decisions to being professionally competitive, across a huge number of tables. Though such an exhaustive performance test of our system was not feasible at the time of its completion, we have gained valuable insight into the overall behavior of our program by analyzing a subset of basic situations. Our system is furthermore heavily dependent on the type and amount of hardware available to be deployed on, so the initial statistics do not necessarily reflect the end-goal performance of our program.

We first looked into implementing a synchronized replication or backup of each space cluster. This would ensure that in the case of a failover in one partition, the system as a whole would continue to run seamlessly (with only a slight decrease in performance). To minimize the risk of total failover, the replicated partitions were placed in separate machines. GigaSpaces automatically attempts to convert the replication into a primary partition while the failed machine is being restored.

After running several tests, using a synchronized replicated schema for our system caused performance to drop on average an order of magnitude. This decrease is not too unexpected, since the replicates were present in machines also running primary partitions. Copying each object across a network synchronously (and preventing other operations from running until replication is complete) would create a bottleneck in performance. In addition, though many of the interactions between services seem trivial, our system would be forced to replicate many of the intermediary objects used to communicate between services.

Another factor that may affect the performance when using synchronized replication is the design of our objects. Some objects, such as a table, contain various complex fields (data structures) that contain information regarding the state of the current hand and all bots associated with the table. When these objects are backed up, GigaSpaces will also have to back up these complex fields. Though we could drastically improve the replication time by modifying the class structures, doing so would create other disadvantages in our program as well.

On the other hand, using an asynchronous replication is not very practical given the fast-paced nature of poker and our platform. Since data regarding a specific hand is very short lived (order of milliseconds), an asynchronous backup would serve no useful purpose other than to store persistent data. At closest, the only useful information to be asynchronously backed up would be certain long-term bot information, such as its chip stack amount.

In any case, we found it most practical to remove the replications completely, since a failover in one partition would only cause a single hand (out of many) to be interrupted. Analyzing the probability of such failover happening in conjunction with that hand being important (one involving a significant pot size) led us to the decision of simply waiting until the machine is restored and restarting that hand. Ideally and with the resources, however, we would likely maintain a synchronized backup to ensure perfect correctness.

Another performance issue we analyzed was based on the time limit we placed for each bot to make a decision. In our current series of tests, bots do not contain a significantly complex algorithm; each decision-making algorithm is fairly quick and would probably not fare well against a decent human player. However, in the ideal case that our system does operate with bots that are competitive, the designers of each bot would want to ideally use up the most amount of time they are provided with to make the best decision.

To replicate this scenario, we incorporated time-stamps within the iterations of services handling bot and table logic and induced an artificial “waiting” time for each bot’s decision-making process. We assume that bots that take longer than the given amount will time out and automatically submit a default action, either a check or fold, as is the case in both live and online poker.

We tested a single 9-player table using a modified SmarterBot through a range of induced waiting times. At higher waiting times (0.5s to 1s), we discovered that the number of hands per hour varied proportionally with the waiting times. In each hand, every bot would be forced to make a “decision” during pre-flop play. On average, 2 to 4 bots chose to take the hand into the flop, and even fewer towards the turn and river. Ignoring overhead costs, this would equate to an average of 10-15 “decision-making” periods per hand. At 1 second per decision, this would yield a 10-15 second period per hand (about 300 hands per hour), and so on. This result is consistent with the typical number of players being involved in most hands in live or online poker.

We gradually reduced the time-limit threshold down to tens of milliseconds, and though the results varied widely, there was an expected increase in the number of hands played. The performance tended to plateau when we reduced the times down to below 25 milliseconds per decision, approaching the same performance as when we did not induce any waiting time. At a combined total of around 100ms per hand, our simulations resulted in slightly under the theoretical average of 36,000 hands per hour. Again, this value varied widely depending on the nature of each hand and the overhead associated with different operations.

As depicted in Figure 8, the number of hands we were able to attain is shown relative to the other alternatives of playing poker, all of which require manual inputs for every decision. During a typical live poker game (such as in a casino), players are able to play around 25 hands per hour. In a single online poker table, this number increases to about 75. Recently, Full Tilt Poker, one of the most popular online sites for poker, released a fast-paced version of a poker table called Rush Poker. Once a player folds a hand, he or she is immediately taken to a new table, reducing the waiting time between each hand. This drastically improves the
number of hands played per hour. However, none of these games come close to the potential frequency achieved by an automated poker platform. We believe that the 36,000 figure is a conservative estimate, and higher numbers can be achieved through an improved system implementation and better hardware.

We note that simulating a waiting time was only feasible because the bots resided in the in-memory grid alongside the services, where communication is much faster compared to a networked communication. In a networked setting where the server communicates with clients to get the bots’ decisions, many other issues arise such as latency which may affect the time limit per decision. In addition, actual performance may be faster without time logging, since the operation to keep track of time uses up system resources in each of the partitions.

7. SHORTCOMINGS

There are several current shortcomings of our project that can be improved in the future. First, our consumer base experiences a very high barrier to entry. That is, in order to be able to participate in our poker games, users must have the capability to program a bot with functionality that is consistent with what we expect. Creating anything at this level of sophistication requires a considerable programming background, and further, making a poker bot that is able to win money requires a sophisticated poker knowledge base as well. Thus, it is difficult to predict the traffic and user base that a platform like ours can attract. However, as both the world of poker become more and more technologically sophisticated, we expect this problem to be less and less of an issue in the future.

Furthermore, our platform is currently lacking any type of graphical user interface (GUI). Because the direct users of our platform will be bots rather than human beings, we decided that the functionality of our system was of primary interest, and therefore did not focus on creating a standalone user interface. Consequently, all results, hand reports, and statistics are currently outputted and analyzed separately. For our platform to be commercially viable, however, we will need to develop some sort of GUI. This way, when users decide to review their bot’s playing history or performance, he or she can do so simply and easily.

Another shortcoming for our automated poker platform is that we do not properly address security issues. Though the platform is designed such that bots only have access to a certain set of information, this does not prevent users from designing malicious bots that could affect the behavior and logic of our system. For example, such a malicious bot may intentionally take a long time to act, delaying the hand and disrupting the game flow. Other bots may attempt to send in faulty decision objects, multiple decision objects, or decision objects out of turn. Although our system generally safeguards against the creation of faulty objects, it has no explicit layer of security or detecting them and thus cannot guarantee to be immune to malicious attacks.

We also do not address the issue of possible collusion that might occur in automated poker. Though collusion exists in any form of poker, allowing bots to automatically make decisions could further amplify the problem. For instance, bots may data mine on the behavior of other bots, and possibly share this information through a separate network or via detectable patterns in the actions it takes.

8. FUTURE WORK

While much of the foundation and basic aspects of the ideal automated poker software have been addressed in our project, there are still many remaining issues that have not been covered.

First, as mentioned before, a graphical user interface must be developed to allow the users and creators of bots to be able to build and maintain their playing algorithms, even if the users do not come from a programming background. All interaction between the client and server is currently done programmatically. Our ideal, long-term goal would be to release a commercialized version of our software, allowing users to play on our product as they do on the many popular poker sites in existence today. Hence, the user interface aspect of our project plays an important part of differentiating our product from others’.

Creating a real-time interface that keeps track of the bots’ performance is not much different from what way we have implemented it already, but can be easily improved with the GigaSpaces platform into a graphical interface as well. Instead of reading the bot objects directly, which affect performance, we can place an additional statistics service in each of the partitions. Clients can then read detailed “stat” objects written directly into the space by these services corresponding to each bot and table. If a client maintains several bots across different partitions, the space proxy would be able to take advantage of the hidden partitioning logic and aggregate all these “stat” objects across all the relevant partitions. Furthermore, by creating a graphical interface, bot managers would be able to track their performance much like quantitative day traders.

We would also need to develop a more practical communication channel between the client and the server components. In our project, the only network communication being used is for the clients to upload its bots into the memory space and to keep track of the bots’ performance. The actual
decision-making process is done via in-memory communication with bots stored directly in the data grid. This presents many security issues in an ideal setting, and we would like the server to instead communicate with client bots through a network. However, the issue of latency rises, which may drastically affect the speed of our platform.

9. CONCLUSION
Although invented almost a century ago, the game of Poker, and especially Texas Hold’em, has never been more popular than it is today. Over the past decade, an increasingly large number of people have begun playing poker online, where more hands are played in a shorter amount of time. We believe that the next natural step in the evolutionary path of poker is automated poker. We developed a prototype of an automated poker platform, where user-designed bots are placed against one another in multiple, high-speed poker games. Our platform is able to effectively increase the number of hands played by several orders of magnitude, therefore reducing the variance of each session and therefore allowing winning players to better realize their expected profits. Furthermore, we developed several sample test bots to assess the functionality and effectiveness of our platform. With a few improvements to our program’s GUI, security, and user-friendliness, we envision our project to be the prototype of the future of online poker.

10. REFERENCES