P2PBay - Peer-to-Peer Auction System

Carlos Ribeiro, João Sampaio, Nuno Nogueira

December 5, 2014

1 Introduction

The goal of this project is to develop a Peer-to-Peer (P2P) solution for a distributed auction system. Besides several possible comands needed in an auctioning application (Sell, Search, Bid ...), we also offer a monitoring solution to measure the number of active nodes, total number of users and items.

2 Problem statement

Current server based solutions for applications lack the capacity for scalability and costs can go skyrocketing if this services are intended to be used for a large number of users.
When for instance the number of users requesting services in an application increases rapidly the server may not be able to reply to all requests or even crash as a consequence.
To solve this problem a solution may be acquiring more servers which costs more money, even if there are periods where the usage is low. In order to reach a solution that overcomes the problem of costs and scalability, in this project we explore the properties of P2P based solutions.

3 Protocol description

3.1 TomP2P

This project uses TomP2P, a distributed hash table (DHT) implementation very similar to Kademlia. Kademlia uses a iterative XOR-based routing that uses a 160 bits uni-dimensional address space and node identifiers are assigned randomly. Due to its iterative routing, it performs parallel searches, allowing the use of the fastest path. There is the concept of k-buckets (called bags in TomP2P): a list of k nodes for each of the sub-trees (or particular bit). TomP2P allows 159 bags with a custom k value (Kademlia uses 20 by default). The keys are located on nodes whose node ID is closest to key, so when searching for a key the lookup is done in parallel for the closest nodes in the neighbor table. The routing takes $O(\log n)$.

3.1.1 Comparision with others DHT

Analizing our system a Kademlia-like DHT seems the best fit. Due to the fact that in our network the peers are users of the system too, we may have to deal with a high churn rate. Kademlia deals well with node joining and leaving, because of the concept of k-buckets, that allows the node to be assigned a k-bucket when contacting a peer in the network, lookup for itself, and subsequently the k-buckets of the other nodes will be updated.

Comparing with DHTs like Pastry that need secondary routing tables (leaf set) for their routing algorithms, the XOR metric and k-bucket concept allows Kademlia to keep it as simple as it is and saves bootstrapping and maintenance cost.

3.1.2 Replication

One of the biggest problems with Kademlia is data maintenance, because nodes with data may leave at any time, so is necessary data republishing and replication. TomP2P allows two types of replication: Direct Replication and Indirect Replication. In Direct Replication, the originator peer is responsible for the data, and needs to periodically refresh replicas. If the originator goes offline the replicas disappear. In Indirect Replication the closest peer is responsible for maintanence of the replicas, and the originator peer may go offline without affecting the data availability. This requires cooperation between responsible peer and originator, and may cause problems with the effective delegation of the responsability for the data. We have choosen to use Indirect Replication due to the fact that goes better with a higher churn rate.
TomP2P has a default value of six in object replication, this means that any object created in one node will be available in a total of six nodes.

3.1.3 Peer join and leave

As our application is implemented using the Kademlia protocol the following events describe the process of a peer joining our peer network:

- For new node U to join the peer network it has to receive an ip address and port of and node W already part of the network;
- Node U adds W to its bag(bucket - list k of neighbors);
- Node U looks up itself, and learns about its neighbors, other peers may also add U to their bags.
- The node closest to each key will store the <key, value> at U if it's one of the closest k nodes.

In our appplication we don't have any dedicated servers: each client in our application helps keeping the DHT. This will produce a higher churn rate as users will login and most likely leave after browsing in the application. This will be mitigated by our choise to use indirect replication.

3.2 Commands

After login, a set of commands are available to the user. This set of commands can be listed by pressing tab. The list of commands seen by the user and its features are the following:

- **statistics**
  User is presented with a set of statistics such as the number of peers and the number of items present in the network.

- **sell**
  By using this command the user can create a new item for sale and replicate it in the network. The operators OR/AND/NOT can be It uses the following syntax:
  sell "name of item" "description of the item" [value]

- **search**
  Presents all the items matching a set of searching words and a logic operator presented by the user. It uses the following syntax:
  search [OR/AND/NOT] "searching words"

- **show-my-items**
  By choosing this command the user can access all his/her items, list his bids, and finalize any of them in order to close the running auction and sell the item. After the list of items is presented the user can chose two different options witch uses the following syntax:
  [Number of item] – lists an item's bid history
  [Number of item] – Finalizes an item (only possible if the item has at least one bid)

- **purchase-history**
  Lists all items purchased by the current user.

- **bidding-history**
  Lists all bids performed by the current user.

- **quit**
  Performs a controlled exit of the network.

3.3 DHT - Information retrieval and storage

In our application we use the DHT managed by TomP2P in order to store and retrieve information generated and searched by the users. The object types referenced in the DHT are: Items, Bids, and Users.
Each object saves a keys reference to other relevant objects to create associations of data types;

For example, an item references an object type List-Bids which contains a set of keys referencing all the bids in the DHT made to that specific item. In order to maintain consistency, a bid object also saves a key reference to the item which is associated with.

The User object references as well all of his/her items in order to manage them.

3.4 Search Algorithm

As any auctioning application, is necessary to have a search command to enable users to search for items beeing sold.

TomP2P allows two types of information storage in the DHT, a simple "put" that simple stores the $< key >, < value >$ and an "add" command that allows append a $< value >$ to a key. This enables a key to have a list of values, we use the "add" command in our search implementation by creating an index structure, where the $< key >$ is a hashed word and the values are all the items identifiers that have that word in their title field.

When we want to search for an item we hash the word parameters in the search query and retrieve the lists of item id that have those words. Then by using operator "AND, OR, NOT" we combine the lists to achive a collection of item ids that satisfy the search parameters and operators.

A search query can be very simple with one operator and two operands, or be more complex and intertwine several simple querys into one large complex one. To allow the use of complex querys we have chosen that the query must be writen using a prefix notation, so that there is no subjectivity.

The pseudo-code of the code we used to implement the evaluation of the query is described bellow:

```
Scan the given prefix expression from right to left
for each symbol
 {
  if operand then
    push onto stack
  if operator then
   {
    operand1=pop stack
    operand2=pop stack
    compute operand1 operator operand2
    push result onto stack
   }
 }
return top of stack as result
```

3.5 Gossip

A gossip protocol is a style of computer-to-computer communication protocol inspired by the form of gossip seen in social networks. By sending periodic messages between peers, complex problems like finding the number of peers in a network is possible with out any centralized server.

3.5.1 Gossip data aggregation

An aggregation type gossip is used to collect and spread information throughout the network in order to calculate a global value of the system. In our case the information will be the number of nodes, registered users and items.

The algoritm we implemented to do our aggregation is the following:

1. Let $(sumR_{node}, sumR_{user}, sumR_{item}, weightR_{node,user,item})$ be all the pairs sent to i in round t-1
2. Let $sumT_{node,t} := sumT_{node,t-1}/2 + \sum_R (sumR_{node})$,
   $sumT_{user,t} := sumT_{user,t-1}/2 + \sum_R (sumR_{user})$,
   $sumT_{item,t} := sumT_{item,t-1}/2 + \sum_R (sumR_{item})$,
   $WeightT_{node,user,item,t} := weightT_{node,user,item,t-1}/2 + \sum_R (weightR_{node,user,item})$
3. Choose a target $f_t(i)$ at random from peer list.
4. Send the pairs $(\frac{1}{2}sumT_{node}, \frac{1}{2}sumT_{user}, \frac{1}{2}sumT_{item}, \frac{1}{2}weightT_{node,user,item})$ to $f_t(i)$ and the other half keep for yourself.
5. The number of nodes estimate is $(\frac{sumT_{node,t}}{WeightT_{node}})$,
   for number of registered users : $(\frac{sumT_{users,t}}{WeightT_{user}})$,
   for number of items : $(\frac{sumT_{item,t}}{WeightT_{item}})$

3.5.2 Gossip message

At the beginning of the application all nodes send every 5 seconds a message to one of their neighours, picked at random.
A gossip message is composed of:

- **Message ID:** used to reset the gossip protocol.
- **Node Sum:** starts at 1 and is used to sum the number of nodes;
- **Node Weight:** stars at 0;
- **User Sum:** starts with the number of users stored in the peer;
- **User Weight:** stars at 0;
- **Item Sum:** starts with the number of users stored in the peer;
- **Item Weight:** stars at 0;

Our application has every one of the above parameters locally stored. Hhen a gossip message is sent, we cut in half the sum and weight values stored localy and send the other half in the gossip message.
When we receive a gossip message we add the values in the message to our local respective values.

3.5.3 Gossip start

One and only one of the peers must be initialized with weight values of 1. This can optimally be done by the peer closest to id 0, but in our implementation a logged user with username ”Admin” is needed to start the gossip manually. This peer will then be responsible for the gossip reset explained below.

3.5.4 Gossip reset

The gossip reset feature enables a long term correction in the information retrieved by the gossip protocol. If a node fails, the calculation becomes compromised, as the “conservation of mass” principal no longer holds thus a reset of current gossip values consensus is required. In our implementation we chose a threshold of 4 minutes which is longer than the convergence time of the gossip protocol, this means that every 4 minutes or so the admin user sends a new gossip message with *newID = previousID + 1* and its initial values. This new message when received by another node “forces” it to discard the old values and to propagate once more its initial values. This allows to maintain an approximate estimative of the real values of the network through out time without concerns of peers leaving.

4 Critical evaluation of the developed protocol, its strengths and weaknesses

To achieve the optimal performance in retrieving information of the DHT a complex analysis of the application structure is needed, which we only performed to a certain level. We believe that by further refactoring our code would be possible to reduce the rate of accesses in the DHT, to make it more efficient.
In the current solution we access the same information several time. By implementing a cache system and changing TomP2P default built-in to retrieve only the objects that have changed, it could be possible to diminish the accesses to the DHT.

Our application allows concurrent accesses to information, and paralel bids on items, as we append bids to items and don’t change the item itself.
If a user searches an item and bids on it, but if the owner finalizes the item inbetween the new bid is appended to the item, then the user will see a success message on the bid but the item will have been finalized and sold to the previous bid user.

To create the management system we use a gossip protocol, because these protocols are highly scalable. we had a problem with our gossip implementation: when a peer leaves normally (without unexpected exit) we send a message to a neighbor with his current sums and weights minus its original values. However not every time the other nodes converge to the new value of nodes, only after a reset do they converge.

5 Experimental evaluation

5.1 Search

For this simulation we started with six nodes and 20 items for sale. The bellow graphic shows the time it took to search the index objects for the words in the query ”search and ipod nano” in the DHT for diffent number of nodes. The measurements were taken by four different nodes after they joined. Each search was repeated five times and then a average value used to create the graphic.

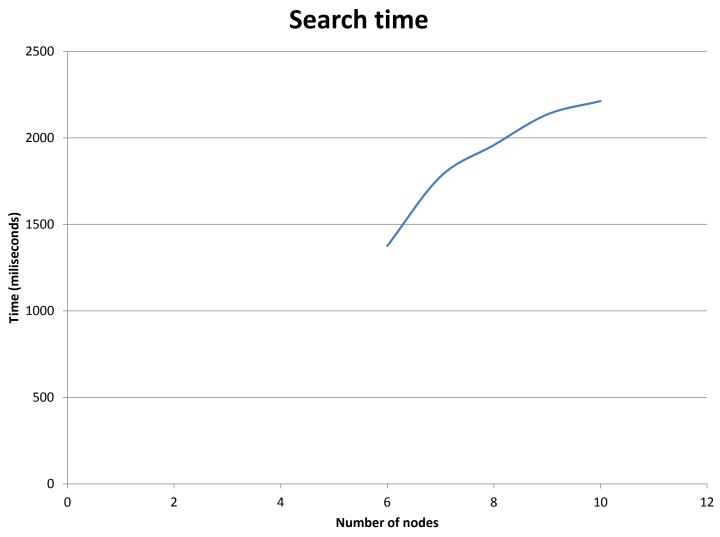


Fig. 1 - Search time of an item for different number of nodes.

We can see a increase in time of search, for six peers, all of then had the 20 items locally. The time spent in search seems to be increasing much more for new peers, so we deduce that the search will converge to a maximum value, but for a reduced number of peers this could not be observed.

5.2 Gossip

5.2.1 Peer join

In this scenario we had a network with five nodes running and then a sixth node joined the network. The following graphics show our results for number of nodes, registered users and items in our application. All these simulations were ran three times and a average set of results were obtained.

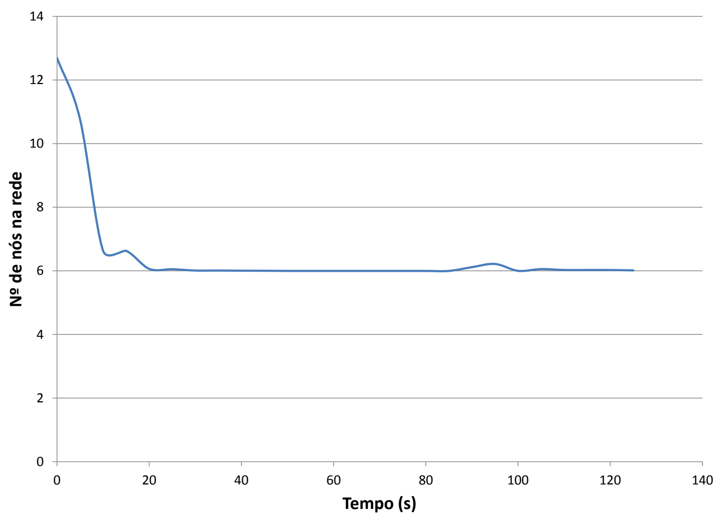


Fig. 2 - Gossip number of nodes observed after peer joined the network.

From the above mesurements we can observe that the time it takes for the number of nodes to converge is roughly 25 seconds.

5.2.2 New user

In this simulation we ran six nodes, then we added a seventh node which then created one new user. The below graphic shows the time since a gossip reset, roughtly 25 seconds after it converged at six registered users, than around second 80 we added the new user and at time 100 seconds it had converged to the correct value of seven registered users.
The observed mesurements were taken by a peer other than the peer that created the new user.

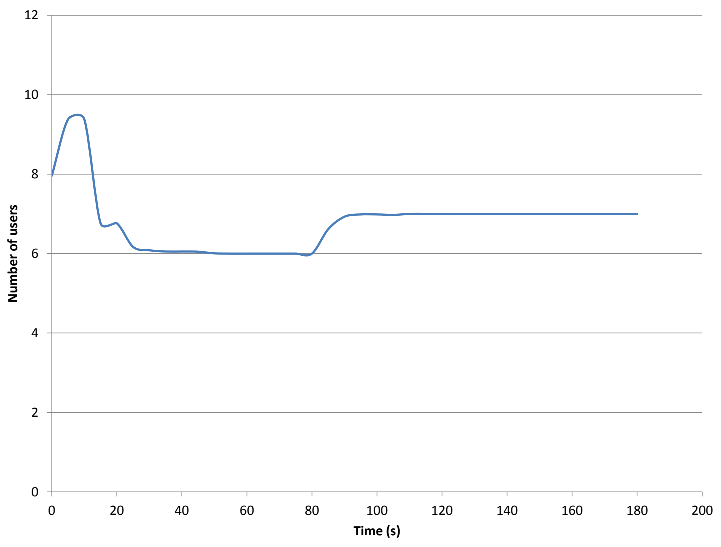
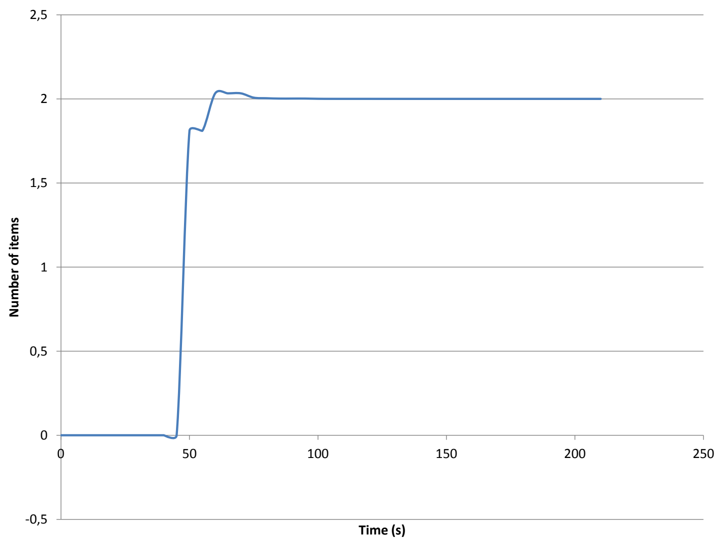


Fig. 3 - Gossip number of registered users, observed by a peer in the network.

5.2.3 New Item

In this simulation we ran six nodes and at 45 seconds since start, two diffent peers added one new item. The below graphic shows the time since a gossip reset. Since the original values were zero the reset is not visible. Around 45 seconds we added the new user and at 75 seconds it had converged to the correct value of two items.
The observed mesurements were taken by a peer other than the peers that created the two new items.



6 Conclusions

In this project we have shown that a pure P2P solution can be considered as a valid alternative for many client-server based applications, with notorious advantages such as lower cost and improved scalability.
Of course this solution is not perfect, P2P decentralized solutions may be vulnerable to several problems, such as security ones, where many different types of attacks can be performed.
Shielding from this treats may not be easy and requires a complete analysis of the solution from many different angles.
Another problem that can be raised is hypothetic situation were many nodes leave all at once, this can be counter measured by replication, but there is always a probability of data loss.
For further testing of the reliability of this system, the next step would be test it with several hundreds or thousands of nodes.