BitTorrent File Sharing in Mobile Ad-hoc Networks[*]

Gábor Balázsfalvi
University of Debrecen, Hungary

János Sztrik
University of Debrecen, Hungary

September 29, 2006

Abstract

This paper presents an application of the Group-based Service Discovery Protocol (GSD) to implement the BitTorrent file sharing protocol on dynamically variable mobile ad-hoc network (MANET) environments. We used a so called 'trackerless' extension of the BitTorrent to make our system more efficient on mobile P2P networks. Also we have implemented a discrete event simulator in Java language, to inquire into detail our protocol. Our simulation results show that proposed GSD with BitTorrent protocol results in shorter download times than the standard file transfer methods.

[*]Research is partially supported by Hungarian Scientific Research Fund-OTKA K60698/2005 and Hungarian Academy of Sciences (HAS)-Korean Science and Engineering Foundation(KOSEF) F01-2006-000-10014-0.

1 Introduction

A mobile ad-hoc network (MANET) [14] comes into existence when mobile devices (e.g. telephones or PDAs) contact each other via a wireless connection, for example Bluetooth. These networks are close to wired P2P networks in the sense that the peers do not have any structure in the layout and each peer can suddenly disconnect from the others. MANETs can be established for example in an office or in a conference room when one wants to share data with others. In [2], Chakraborty et al. proposed a method called Group-based Service Discovery for presenting and discovering services in MANETs. That protocol is able to speed up the discovery of a shared file, but doesn't speed up the downloading process of a large file. In [10], Sai Ho Kwok showed that the large video and audio files have been transferred among P2P users and the mobile devices which are able to play video data, and have Bluetooth connection have become very popular. The users in wired P2P network environments share their data mainly by the BitTorrent protocol [4, 8]. This protocol is applied because it speeds up the large-scale file sharing, as Bharambe et al. have analyzed in [1]. We used the GSD service discovery protocol to introduce a so called 'trackerless' implementation of the BitTorrent protocol in mobile ad-hoc networks. A simulator has been developed to find out whether the proposed method is better than the standard one with respect to the downloading time.
The remainder of this article is organized as follows. In section 2, a review of the GSD and the BitTorrent protocols is given. In section 3, we propose a protocol of ad-hoc file sharing. In section 4, simulation results illustrate the comparisons versus simple data download approach. The paper ends with a summary and conclusions.

2 Recent works

2.1 The Group-based Service Discovery protocol

In [2, 3] Chakraborty et al. have presented the GSD protocol, which is a service discovery protocol for ad-hoc networks, based on three concepts:

1. Bounded advertising of services in the vicinity;

2. Peer-to-Peer dynamic caching of service advertisements;

3. Service group-based selective forwarding of discovery requests.

The first one means that peers continually advertising the services they provide, such as sharing a file. Bounded advertising means that there is a hop-limit on each advertisement. Every peer that forwards an advertisement decreases the hop-counter, and when it becomes negative, the advertisement passes away. The second concept means that the arrived advertisements get cached at peers. Aside from advertising their own services, the peers also advertise the so-called groups of services they have seen. GSD uses a hierarchical grouping of services [6]. This is one of the many semantic based service discovery approaches. Others are described in [11, 7]. We chose this because the data sharing services can be grouped in many ways, e.g. by content. The GSD protocol, when the services are well grouped, is very efficient comparing to other approaches. It is shown to be fast and does not load the clients too much.
Roughly speaking it works as follows. Every node in the MANET periodically sends a list of its own services within the radio range. This advertisement contains information about the services of the sender and their groups. Also it contains a field of containing groups which were seen by the sender. The redundancy of advertisements is avoided by a broadcast ID that is monotonically increased with every broadcast message, and a hop-count value, that has to be decremented after sending an advertisement farther. If the advertisement a node received was not redundant, before forwarding, the receiver extracts it and stores information in a service cache. It decreases the hop-count, and in the case latter is greater than zero, forwards the advertisement. The package also has the lifetime of each service advertised. They only should be cached until this expires. When a node needs a service, it first checks its own cache. If in the advertisement range there is such a service advertised, in an ideal case, the cache should contain an entry about it. If not, it sends a broadcast message to nodes in the vicinity, with the name and group of the service and a maximal hop-count value. If the node receiving this query has already seen that group, it forwards selectively the message. Otherwise it sends to every node in the radio range the message. Before forwarding, the hop-count must be decreased. If a node founds the service, it answers the query. Otherwise the service is not provided in the hop-count range.

2.2 The BitTorrent file sharing protocol

In [4] Cohen has presented a P2P file sharing protocol called BitTorrent which is based on the up-link capacity of the down-loaders. There must be a server somewhere at a well known place which is used to coordinate the others. This server is called tracker. From the data to be shared has to be split into several pieces and a torrent file must be created and uploaded to the tracker. The torrent file contains hash and size information about the pieces, their length, name of the original files, and the URL of a tracker. When another peer wants to get the shared data, it has to contact the tracker for a list of peers downloading or uploading the same data and having the same torrent file. Peers that have the full data and are only uploading are called seeders, while the peers that are downloading as well are called leeches. The health or visibility of a torrent shows if the file was fully reachable or not. It is the number of full copies of the data in the system. Every seeder counts one in this number. A leech having e.g. 30% of the data which no other leech has counts 0.3 in the health value. So, when this value is below one, the data is not available. If there are seeders in the system, then the health is at least equal to their number, and the data can be downloaded. The peer tries to connect to all of the peers present in the list it got from the tracker, namely its neighbors. Each peer reports to all of its neighbors what pieces it has and also tries to download from its neighbors pieces that it does not have. Except for the very first and the last pieces, a peer selects the piece for downloading the piece that is the rarest one among its neighbors. The first one is selected randomly, and the last ones are selected by the fastest connection. This is the case because, at the beginning that peer needs a whole piece of the data as quickly as possible to be able to upload. The leeches give the upload bandwidth in a tit-for-tat manner. The more they get from one peer the more they give. Leeches periodically disallow the downloading from some connections. This is called choking that peer. Peers are then unchoked in the order of the upload bandwidth, except for one, that is randomly chosen to be able to detect faster connections. This is called optimistic unchoke. Because seeders usually leave the system after a share ratio of one - that is after they uploaded as much as they downloaded - they prefer peers with a better connection. These are able to share the data as fast as possible. Measurements have shown that the BitTorrent scales well and it is efficient [5, 15].
Recently there are some clients that use an extension of this standard system.

They use a distributed hash table to store the torrents instead of a fixed tracker. Although there is no standard implementation, some clients are using the Kademlia [12] system. The distributed hash table is based on the followings. Every participating node gets a unique random integer value of the same size, representing a key from a so called key space. By these keys, the nodes form an overlay network. A typical arrangement of them can be a binary tree allowing logarithmic time search among the keys. The nodes have a common method for calculating a so called distance between two keys. When a node creates a torrent, it makes a hash value of that. Every node has to use the same hashing method, e.g. SHA1. This method gives an integer having the same size as the IDs. Next the creator looks for the node having the closest key to the hash value. When a node wants to find that torrent, it has to compute the hash value of it, and look for the node having the closest key. That node has information regarding the peers using the same torrent. This information has to be updated with the address of the new peer as well.

3 Service discovery based BitTorrent

Our file sharing solution in MANETs is based on group based service discovery and distributed hash table based BitTorrent. The idea is to introduce groups for different kinds of data to be shared, for example movies, music or picture. All of them are grouped together into one main group, e.g. the 'Share' group, henceforth, denoted by 'S'. This is the only group we have modeled actually for the sharing services, and a concrete file to be shared uses the service $[S : "file"]$. We use another group of services to assign the nodes participating in the distributed hash table system. We assume that every peer taking part in the hash table provides a service in group 'D'. There may be several other kinds of distributed hash tables, but only one of them is for data sharing, namely $[D : S]$. When a peer has data to share, it has to create the torrent and put its own address into it. Next with the help of GSD service query, it looks for a provider of $[D : S]$. After it knows at least one node from the hash table, can easily look for the right place for the torrent data by hashing the torrent file. When another peer needs the data, first it uses the GSD system to find a peer seeding or leeching the same data. For this, the searcher does not have to know the exact name of the torrent, because the GSD system can answer the query with every related torrent. The searcher selects a torrent, creates the hash of it, and by looking in the hash table it can find all the peers sharing the same data. The searcher has also to register itself into the hash table. For this, it uses again the GSD system to search for a node joining in the distributed hash table. Next we present the algorithm for data sharing:

Algorithm 3.1: SHARE(A, "*file*")

```
# Preparing "file" to share. 'A' is an owner and it provides [S : "file"].
$T \leftarrow GSD\_look\_for$([D:S]);
if $empty(T)$
  then end with fail;
$h \leftarrow hash(torrent("file"))$;
$N \leftarrow find\_DTH\_node(T, h)$;
$put\_into\_DHT(N, h, torrent("file"))$;
```

Here T is a set of nodes, all providing $[D : S]$, the distributed hash table for data sharing. 'A' has to create the hash value of the "file"'s torrent and store it in h. Next it uses T and h to find N, the node that is responsible for data with key h. Finally 'A' stores the created torrent in the hash table. The next algorithm describes the needs for downloading "file". It is important, that a node, looking for the data, is not able to create the torrent file, because for example, it does not know the number of pieces.

Algorithm 3.2: SEARCH(A, "*file*")

```
# Looking for shared "file". 'A' is a peer.
$S \leftarrow GSD\_look\_for$([S:"file"]);
if ($S$)
  then end with fail;
$t \leftarrow select\_torrent(S)$;
$T \leftarrow GSD\_look\_for$([D:S]);
if $empty(T)$
  then end with fail;
$h \leftarrow hash(t)$;
$N \leftarrow find\_DTH\_node(T, h)$;
$R \leftarrow get\_from\_DHT(N, h)$;
$R \leftarrow R \cup \{A\}$;
$register(t, R)$;
$put\_into\_DHT(N, h, t)$;
# Start download using R as a list of peers
```

First 'A' gets a set S of nodes having the data "file" from the GSD system. It could also be a group of data, 'conference video' for example. Next 'A' selects the torrent t it wants to use for downloading. Although, this torrent might have a list of some peers already, this list might not be complete. After that 'A' looks for a node in the hash table, and if does not find any, it has to finish. Otherwise it creates the hash value of the selected torrent, and stores it in h. Next 'A' finds the node N storing the list R of all peers concerning the torrent t, adds itself into this list, registers in the torrent, and refreshes the torrent at the node N. Finally it can use the BitTorrent algorithm to download the data.

4 Experimental results

4.1 The discrete event simulator

This section presents an event oriented discrete event simulator developed to model the proposed protocol. We have built an event and process oriented discrete event simulator 1 using the Java language, mainly as described by

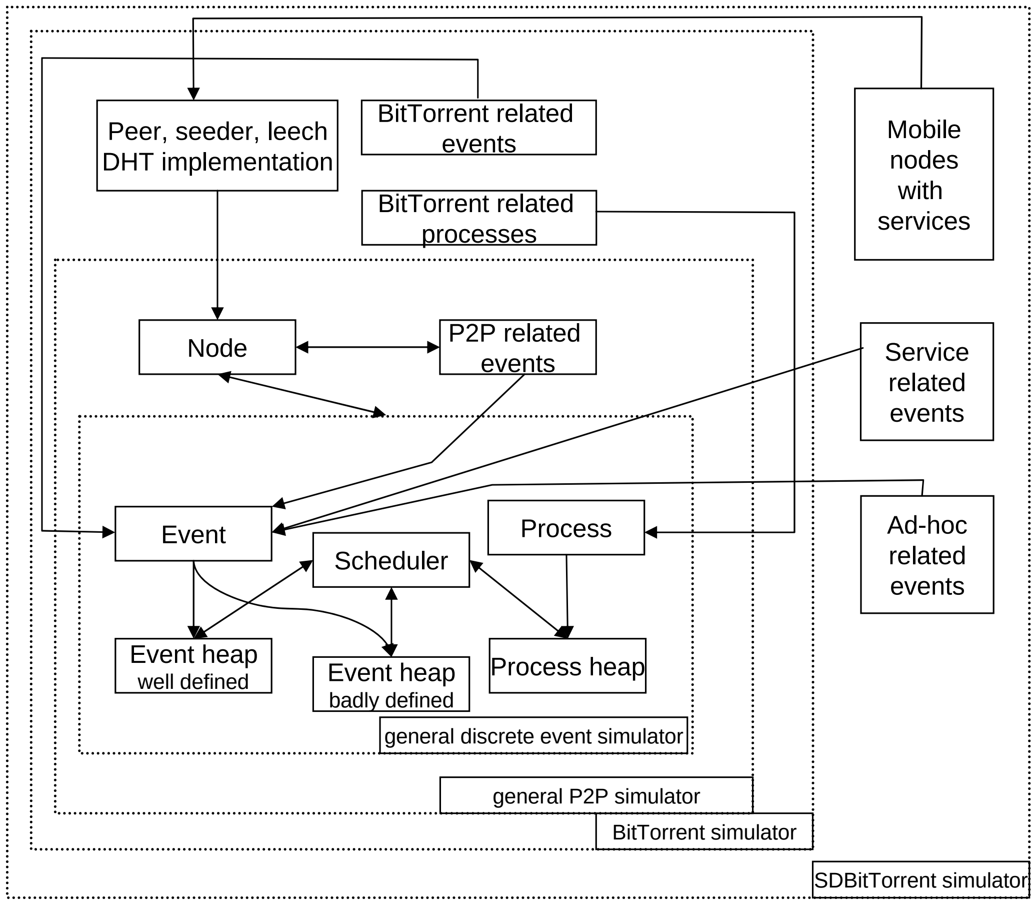

Figure 1: Simulator components

Jain in [9]. The main component of it is a scheduler, which is capable of scheduling events at a specific time, events after a specific time, and processes having the finishing time varying dynamically. For these, it uses three heaps ensuring in logarithmic time the pick up of the event or the process having the turn. The *Event* class is the ancestor of every event in our system. It has a *run()* method, in which it does its work as described later. It can be scheduled at a specific time (well-defined schedule-time) or sometime after a specific time. The latter is called badly-defined schedule-time event, because its schedule time is determined by other events dynamically. We can only say that the event does not fire before a given time, but it fires before the next well-defined schedule-time event having a latter schedule-time. The

other kind of items that can be scheduled in our system are processes. The ancestor of every process is the *Process* class. This class has a *finishTime()* method that is invoked at every related event's fire-time and every related processes final time with an actual parameter of the current time. After that, these processes can do their own work which has to be done between the last call and now. They must prognosticate their final time with the actual conditions. These conditions can change only by other processes or by events, so the correct finishing time cannot be missed.
To model a peer to peer system, we need node objects and events, to create these nodes (*BornEvent*) and also to destroy them (*DeathEvent*). Also we have to model an abstract connection between peers.
For the BitTorrent system to be modeled we have to derive the peers from the more general nodes. Also we have to let a peer be a seeder or a leech. These nodes are also responsible for the distributed hash table. We modeled roughly the Kademlia system as described in [12]. After a peer is born, a *GetPeerListEvent* object responsible for the neighbors of one peer is scheduled. It can schedule another *GetPeerListEvent* at a later time, because the peer may need to renew its neighbors, for example when a neighbor becomes seeder. It also schedules a *SelectPieceEvent* object. This object implements the different kinds of piece selection policy (the rarest first one for example). If piece is available for download, then it schedules a *SelectPieceEvent* after the next event's time. Otherwise it schedules an *UpdateLinksEvent* object. This creates and refreshes the download processes. When every piece is downloaded, a *BecomeSeederEvent* runs and the peer is turned into a seeder.
In the recent system, the only process class is the downloading of a piece. Its finish time depends on the actual bandwidth, for which it uses a *Link* class.
For the ad-hoc network environment we needed mobile nodes being able to have and advertise services. We added coordinates for the nodes representing the physical place in the MANET of that. A node can only contact other nodes in a given circle, as normally in MANETs. Certainly it can download from a remote node if there is a chain of nodes between them. We implemented roughly the Ad-hoc On-Demand Distance Vector (AODV) routing protocol described in [13]. Its main advantage is that it builds the links between two nodes just in time, and also is able to select a good way for the data which is sent. When a node 'A' needs a connection to another node, 'B' for example, it broadcasts a message for the need of this connection. If 'B' is in the radio range, it answers back. If it is not, 'B' will be found by other AODV nodes, by forwarding the original message. If one tells about 'B' with

a chain, (firstly 'B' does this with only itself in the chain), it appends itself to the beginning of this chain, and answers the message. Of course message loops are avoided. A service is described by a subclass of the *Service* class. It has name and group fields and *preReq()* method for determining the peers that can use the service. When a peer starts using the service the system calls the *useService()* method. When a service discovery request is unsuccessful, the *onFail()* method is invoked. Every node is able to discover other nodes. This is handled by the *DiscoveryEvent* class. Nodes discover services by the *ServiceDiscoveryEvent* class.
We implemented the algorithms described in the previous section in our system with the combination of the two protocols described above. We used two subclasses of *Service* class, one for the distributed hash table and one for the file-sharing service. The provider of a file-sharing service can be a seeder or a leech. The simulation can be displayed graphically with the Java's swing components.

4.2 Results versus plain download

We have run several scenarios with our system. The result of these simulations can be seen in Figure 2. We set up typical file size and bandwidth values. The size of the file to be shared was set to 4 Megabytes, which is typical size of a mp3 music file. Both the uploading and the downloading bandwidths were set to 32 Kilobytes/sec. With these properties a simple download of the file takes 128 seconds. There is only 1 initial owner of that file. In this Figure we can see one continuous line showing our results given by our algorithm. From the two dotted lines, one of them shows the theoretical values of the optimal plain download. This can be easily computed by the following reasoning. Let us start with just 1 downloader. It needs 128 seconds to finish its download. Then two other peers can download the file simultaneously and it takes another 128 seconds, so they finish by the 256^{th} second. If these 3 peers wanted to download earlier, at the 0^{th} second, the download time would be $128 * 3 = 384$ seconds, so the previous was optimal. Continuing this reasoning, we find that if we had $K - 1$ peers which wanted that file and only 1 peer which owned it, we would need $128 * (\lceil log_2(K) \rceil)$. E.g. for 66 peers and 1 owner the K was 67 and the time needed is $128 * 7 = 896$. The peers have to find each other, which certainly also takes time, and the probability that every peer connects to the system at the exact time is very small. The worst case is when every peer wants to download the file at the

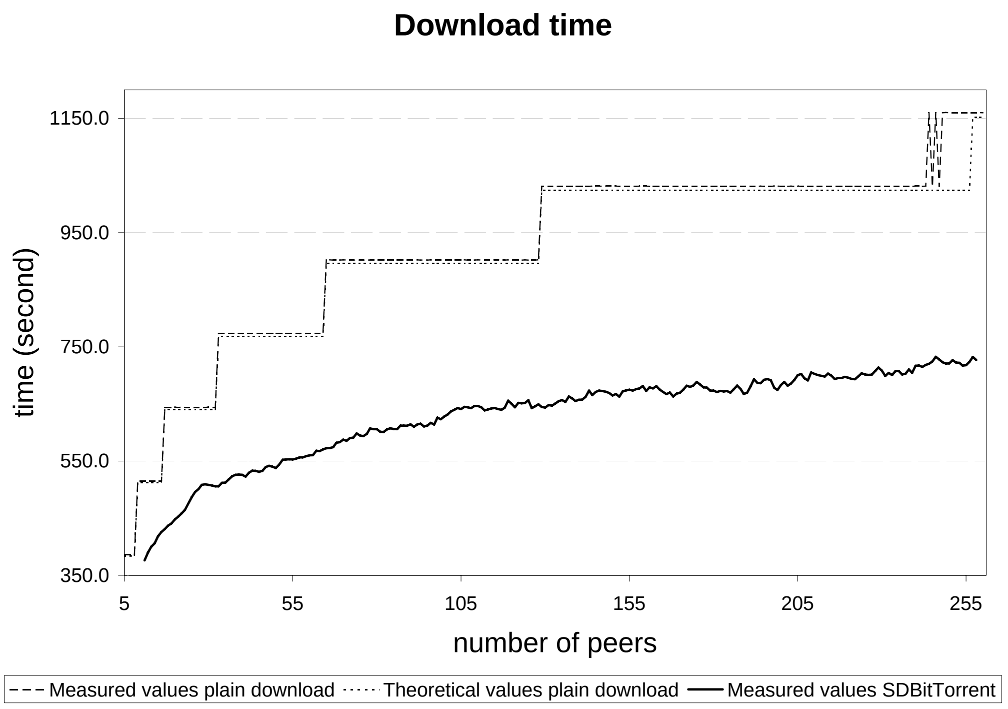

Figure 2: SDBitTorrent vs. plain download

same time e.g. at the 0^{th} second. Then it takes $128 * 66 = 8448$ seconds plus the time to find the owner by GSD. The optimal case of the simple download can be modeled by a BitTorrent system with the following parameters. The number of the pieces of the file is only 1. The maximum number of simultaneous down-links and up-links per node are both 1. We run a scenario with these parameters while the number of all peers was varying between 5 and 260. The second dotted line shows the results of this scenario.
The continuous line belongs to the average of 100 independent simulation scenarios. We have run scenarios for this situation with the following additional parameters. The number of pieces of the file was 100. The maximum number of one's neighbors was set to 20, and the maximum number of simultaneous down-links per node was 10. The maximum number of simultaneous up-links per node was 5. The lifetime of an advertisement was 100 seconds. E.g. when the number of peers was 67 the average sharing time was 565 seconds, which is much less than the optimal 896 seconds of the normal download process.

The more peers we have, the better are the results.

5 Conclusions

We presented a method that can be used by mobile devices in a mobile ad-hoc network, to share data using the BitTorrent protocol. For this we used the Group based Service Discovery protocol and the Kademlia distributed hash table system. Java-based software simulator was developed for simulating the details of the underlying ad-hoc network and the proposed algorithms. By running several experiments we showed that our protocol scales well with the number of peers. The package was able to simulate not just our protocol, but the optimal plain download process, too. We illustrated that our new protocol, containing these setups is much better with respect to the downloading time than the old-style protocol for download.

References

[1] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving bittorrent performance. Technical Report MSR-TR-2005-03, Microsoft Research, 2005.

[2] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha. Gsd: a novel group-based service discovery protocol for manets. In *Proceedings of the 4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN'02)*, Stockholm, Sweden, 2002.

[3] D. Chakraborty, A. Joshi, and Y. Yesha. Integrating service discovery with routing and session management for ad-hoc networks. *Ad Hoc Networks*, 4, 2006.

[4] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[5] D.H.J. Epema, J.A. Pouwelse, P. Garbacki, and H.J. Sips. The bittorrent p2p file sharing system: Measurements and analysis. In *Proceedings of the 54th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, USA, 2005.

[6] Z. Gao, L. Wang, M. Yang, and X. Yang. Cnpgsdp: An efficient group-based service discovery protocol for manets. *Computer Networks*, in Press, 2006.

[7] R. Heckel, A. Cherchago, and M. Lohmann. A formal approach to service specification and matching based on graph transformation. *Electronic Notes in Theoretical Computer Science*, 105, 2004.

[8] M. Izal, Urvoy-Keller G., E. W. Biersack, P. Felber, A. Al Hamra, and L. Garc'es-Erice. Dissecting bittorrent: Five months in a torrent's lifetime. In *Proceedings of the 5th Passive and Active Measurement Workshop*, 2004.

[9] R. Jain. *The art of computer systems perfomance analysis.* John Wiley & Sons, New York, 1991.

[10] S. H. Kwok. P2p searching trends: 2002-2004. *Information Processing and Management*, 42, 2006.

[11] S. A. Ludwig and S. M. S. Reyhani. Introduction of semantic matchmaking to grid computing. *Journal of Parallel and Distributed Computing*, 65, 2005.

[12] P Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *First International Workshop on Peer-to-peer Systems*, 2002.

[13] C. Perkins, E. Royer, and Das S. Ad hoc on-demand distance vector (aodv) routing. *RFC*, (3561), 2003.

[14] F. Zambonellia, M-P. Gleizes, M. Mameia, and R. Tolksdorf. Spray computers: Explorations in self-organization. *Pervasive and Mobile Computing*, 1, 2005.

[15] G. Zihui, D. R. Figueiredo, S. Jaiswal, J. Kurose, and D. Towsley. Modeling peer-peer file sharing systems. In *Proceedings of Infocom03*, 2003.