Building a modular BitTorrent model for ns-3

Elias Weingärtner[1], René Glebke[1], Martin Lang[2], and Klaus Wehrle[1]

[1]Communications and Distributed Systems, RWTH Aachen University

[2]Logic and Theory of Discrete Systems, RWTH Aachen University

[1]Email: firstname.lastname@comsys.rwth-aachen.de
[2]Email: martin.lang@rwth-aachen.de

ABSTRACT

Over the past decade BitTorrent has established itself as the virtual standard for P2P file sharing in the Internet. However, it is currently not possible to investigate BitTorrent with ns-3 due to the unavailability of an according application model. In this paper we eliminate this burden. We present a highly modular BitTorrent model which allows for the easy simulation of different BitTorrent systems such as file sharing as well as present and future BitTorrent-based Video-on-Demand systems.

Categories and Subject Descriptors

C2.2 [**Computer Communication Networks**]: Network Protocols; I6.8 [**Simulation and Modeling**]: Discrete event simulation

Keywords

ns-3, BitTorrent, P2P networks

1. INTRODUCTION

The usefulness of any network simulator grows with every protocol and communication system it is able to model. ns-3 has significantly progressed lately in this regard, as it now provides models for a rich set of protocols at all layers of the protocol stack. However, regarding application layer models, one of the most prominent protocols in the Internet eco-system is missing: BitTorrent.

BitTorrent [5] is a P2P system originally designed for effectively sharing large amounts of data over the Internet. Over the past years, many extensions and changes to the original BitTorrent protocol have been proposed and implemented into client software. They range from performance improvements, such as Super-seeding [8], to tracker-less operation that allows for a fully decentralized sharing of data using a DHT [10]. More lately, adapted BitTorrent clients have been proposed in the literature that alter the piece trading strategy, for example in order to improve the own downloading performance [16] or to create Video-on-Demand services on top of the BitTorrent protocol [20].

This variety in existing BitTorrent systems shows that an according model for ns-3 should not only model a specific BitTorrent client, but instead should allow for the easy replication of the behavior of different BitTorrent systems. This is the main objective of this paper, whose specific contributions are the following:

- In Section 3 we present the high-level design of our ns-3 BitTorrent model. The design is highly modular and allows for the easy implementation of different BitTorrent systems, ranging from common file sharing applications to BitTorrent-based Video-on-Demand (VOD) systems.
- We discuss a BitTorrent model for ns-3 (Section 4) that we have implemented based on these conceptual considerations. It not only features the modular recreation of different BitTorrent systems, but also facilitates the easy scripting of the BT simulation's progression using so-called story files.
- We evaluate the BitTorrent model using a set of micro-benchmarks (Section 5). We also show that our simulation model provides a good modeling accuracy of common BitTorrent file sharing for most parameter settings.

We also discuss the current limitations of the model in Section 6. In Section 7 we compare our model with existing BitTorrent simulation models and simulators. Section 8 concludes the paper with final remarks.

2. BITTORRENT: A BRIEF OVERVIEW

BitTorrent [5] (BT) is a Peer-to-Peer (P2P) system for the distribution of bulk data such as multimedia content or software installation images. A BitTorrent swarm consists of two kinds of parties, a set of BitTorrent clients and the BitTorrent tracker.

The *BitTorrent clients* implement the BT protocol [1] that carries out the data exchange between the nodes. In the following, we refer to this TCP-based protocol as *Peer wire protocol (PWP)*. BitTorrent divides the shared data into small *pieces* of data. The length of a piece typically ranges between from 128KB to one MB. In order to start downloading data from a BitTorrent swarm, the client is supplied

WNS3 2012, March 23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247722

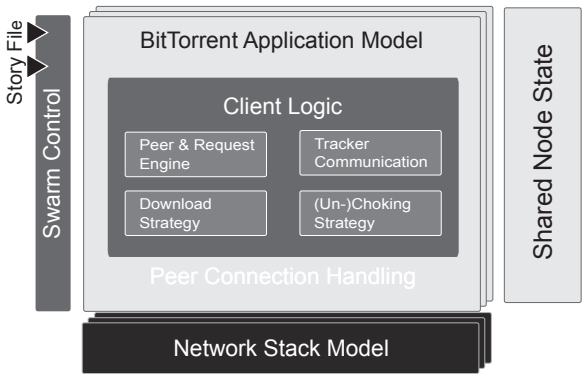


Figure 1: Conceptual overview of our BitTorrent client model: All distinctive features of a BitTorrent client are encapsulated in functional blocks, allowing for a flexible adaption of the client behavior.

with a .torrent meta-information file that contains a list of all pieces. The .torrent file also contains a SHA-1 hash value for every piece in order to allow the BitTorrent clients to verify the integrity of the pieces it retrieves. All BitTorrent clients may request any piece from all other peers in the swarm. Before requesting actual data, a BT client informs the remote client of this intent by sending a so-called *interest* message. Besides the mere exchange of data, the BT clients also exchange messages among each other to notify other BT clients about the pieces of data they posses. A very important particularity of BitTorrent is the so-called "choking" principle. After having expressed the interest in downloading a piece, a BT client may be first blocked (choked) from downloading by the remote peer. This allows a BT client to implement a "tit-for-tat" strategy that favors clients from which remote peer has already retrieved data. To avoid deadlock situations, the clients periodically *optimistically unchoke* other peers and hence allow them to request data without any prior data transfer. It is noteworthy that, although the message formats used by the PWP are well described in the protocol description [1], there is no exact specification for the algorithms of the choking/unchoking behavior or the piece selection strategy. Hence, all available BitTorrent client implementations exhibit a different behavior in this regard.

The so-called *BitTorrent tracker* centrally observes a BitTorrent swarm. It keeps track of all clients that are sharing data in the swarm and provides new peers with a list of BitTorrent clients that are also participating in the swarm. The so-called *tracker protocol* that allows clients to retrieve a list of nodes in the swarm is built on top of HTTP and solely uses GET requests.

3. CONCEPTUAL DESIGN

Corresponding to the architecture of BitTorrent, our ns-3 model in fact consists of two models, a BitTorrent client model and a BitTorrent tracker model.

3.1 BitTorrent Client Model

Figure 1 displays a high-level architecture of our BitTorrent client model. The model consists of the core BitTorrent model that sits on top of the network simulator's network stack model and two singletons that govern the *swarm control* and hold the *shared node state*.

3.1.1 Swarm Control

The swarm control component is responsible for the the configuration and the execution of a BitTorrent simulation. For this purpose, it parses a so-called *story file*. The story file consists of a set of commands in a domain-specific language (DSL), allowing for specifying the simulation set-up and the activity of the BitTorrent clients during the simulation run. Hence, the user of the BitTorrent simulation uses the story file for example to instruct a set of nodes to start requesting a file at a certain point in time. Other commands in the story file allow one to configure the piece selection strategy used by the clients, the user to set the initial distribution of pieces in the swarm or to hand over a .torrent meta-info file to the simulation.

3.1.2 Shared Node State

A major design goal of our BitTorrent simulation model is to enable large-scale BitTorrent simulations with hundreds or thousands of simulated clients. In order to achieve this goal we reduce the individual state space required by each node by storing redundant node state information only once. The most significant state information in this regard is the file shared in the swarm that may range from a few megabytes to a couple of gigabytes. While using an actual payload would not be required for a pure BitTorrent simulation, we decided to use real-world payloads to enable emulation studies with BitTorrent at a later point in time. Storing the data payload at one central component dramatically reduces the memory footprint of one simulated node and hence the memory requirements of the entire simulation. We later evaluate the memory requirements of our model at greater detail.

3.1.3 BitTorrent Application Model

The BitTorrent client model aims at reproducing different BitTorrent systems and software clients. For this reason we abstained from "hacking" together a monolithic BitTorrent client and rather designed the BitTorrent model in a highly modular fashion, in which compositions of different functional units define the actual client logic of the model.

Peer & Request Engine.

Its core task is to translate the high information specified in the supplied .torrent file and the behavior specified by the story file into the logic of the model. It keeps track of the pieces a client has retrieved and provides this information to the other functional units. The peer and request engine also maintains state information for the peers a client is aware of and if these clients are choked, or an interest message has been sent to them.

Tracker Communication.

The tracker communication unit carries out all interaction with the BitTorrent tracker, mostly in order to obtain a number of peers that participate in the swarm. For the actual communication, the tracker communication unit encapsulates a HTTP client and adequate sub-components to parse the meta-information it retrieves from the tracker.

Download Strategy.

The download strategy unit controls which parts of the file are downloaded next from other BitTorrent clients. Classic BitTorrent file sharing clients typically use a *rarest-first*

strategy. As the global availability of the pieces is not known, the clients count how many of the peers they are connected to possess each piece. This information is then used to request the least popular pieces with the goal of maximizing the availability of each piece.

The encapsulation of the download strategy into a own logical unit is important to support other schemes in our model: The *pure sequential* strategy downloads the parts of the file one after another. This scheme is a very naive strategy for the delivery of streaming media. We are currently also in the progress of supporting more sophisticated VOD strategies, for example Give-To-Get [12].

We do not schedule the requests at the piece level, but at the *block level* of BitTorrent. BitTorrent internally splits a piece into a set of a few blocks in the magnitude of tens of kilobytes. By operating at the block level, we can retrieve a piece from several peers and thus are able to avoid problems if the remote peer goes offline while a client is still downloading a piece from that peer.

Choking/Unchoking Strategy.

The choking/unchoking strategy is responsible for blocking/unblocking other clients from downloading pieces. In the classic BT file-sharing service this decision is largely dependent on the amount of data a remote peer has already sent to the client, resulting in a *tit-for-that* trading scheme. However, the modularization of the choking/unchoking behavior allows for arbitrary strategies to be implemented, like randomly (un)-choking remote peers or basing this decision on recommendations from other peers.

Peer Connection Handling.

This functional unit implements the Peer Wire Protocol (PWP) and exchanges both payload data as well as status messages with other peers. It uses the socket layer abstraction of the underlying network simulator - in our case ns-3.

3.2 BitTorrent Tracker Model

In addition to the client model, we also created a rather straightforward BitTorrent tracker model. Once a node joins the swarm, it registers at the tracker. The BT tracker model stores the client information, most importantly its IP address and the peer ID, in a local data structure. It then sends a tracker response to the client, which, among different status information, contains a list with the IDs of other peers in the swarm, their IP addresses and the TCP ports at which they are listening. As all communication tasks are handled using HTTP, our tracker models also incorporates a respective HTTP sub-component. This sub-component provides basic mechanisms for sending GET requests and for parsing HTTP headers; it provides its functionality to upper layers using according callbacks. In the following, we omit a further discussion of the tracker model's implementation due to its simplicity and because of the space constraints of this paper.

4. CLIENT MODEL IMPLEMENTATION

Figure 2 gives an overview of the implementation structure and the component interaction of our BitTorrent client model. Its core is formed by the `BitTorrentClient` class, which is inherited from the `Application` class of ns-3. A BitTorrent client in the simulation is represented by one instance of this class. It stores information about the current client state and acts as a dispatcher among the different components of the client. First, there is the `Peer` class. This class represents an interface to a remote BitTorrent client. It implements the real-world network representation of the BitTorrent protocol and encapsulates an ns-3 TCP socket for communication with a remote client. All data sent out via the `Peer` class is binary compatible with the BitTorrent protocol specification. This allows hybrid evaluation in emulated networks with existing BitTorrent software. The control logic of a (BitTorrent-based) protocol is captured in several strategies. The concrete strategies are implemented in classes which are inherited from the class `AbstractStrategy`. A protocol implementation usually consists of several strategies each of them covering a dedicated aspect of the protocol logic. This approach allows us to easily adapt the simulation to the variety of possible strategies for choking, piece selection and neighbor discovery (tracker- and DHT-based). Hence, we are able to compare the performance of different protocols with low implementation effort.

4.1 Component Interaction

A `BitTorrentClient` object represents one instance of a BitTorrent client in the simulation. It stores the status information of the client such as already received pieces, connections to remote clients, and information on the shared file and the swarm. Furthermore, the client object coordinates the communication between the network implementation of the protocol in the `Peer` objects and the protocol logic implementation in the strategies.

The structure of the program flow and component interaction was strongly influenced by the event-based paradigm of the ns-3 simulator. We introduced client-internal events for all network interactions of the BitTorrent protocol, such as reception of a choking message or the completion of the upload of a requested piece. The events are generated by `Peer` objects and passed on to the `BitTorrentClient` object. The `BitTorrentClient` class implements an event dispatcher infrastructure based on ns-3 `Callback`s. Strategies can register for certain events and are notified by the client class upon their occurrence. This allows strategies to monitor the subset of the current state of the client they need for operation without having to implement a polling policy for state changes.

The communication from the strategies to the client and the `Peer` objects is realized by direct method invocation. We intended the peer and client classes to provide the full set of communication subroutines needed and the protocol logic to fully reside within the strategy classes. Hence, we saw no benefit in using an event-based approach here. The less number of indirections increases the readability and speed of the code. Following the spirit of ns-3, all requests are handled internally in an asynchronous first-come-first-served fashion, while simulation time is guaranteed to be advanced only after all event listeners have been given time to react. Our model hence regards a client node's computational effort as zero for all operations.

4.2 Strategy Implementation

We decomposed the logic of the original BitTorrent protocol as given in [1] into four aspects: part selection, choking/unchoking, peer connection choice, and request scheduling. The part selection strategy determines which parts

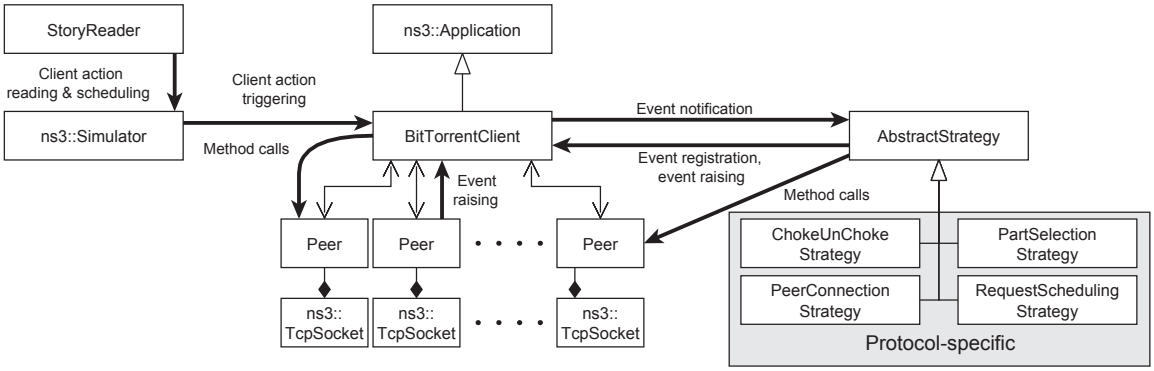


Figure 2: Class collaboration in our BitTorrent client model: The `BitTorrentClient` class owns a set of `Peer` class instances of which each provides an interface to one remote peer via a TCP connection. A set of derivatives of the `AbstractStrategy` class forms the BitTorrent application's logic. The `Peer` class announces any Peer Wire protocol events to the strategies via an event notification mechanism provided by the `BitTorrentClient` class, whilst communication with remote peers can be initiated directly through method calls with the `Peer` class. For an scripting of scenarios, the `StoryReader` interprets a DSL and schedules actions of selected clients.

of the shared file should be acquired next and from which peer. The choking/unchoking strategy decides which remote clients are allowed to request pieces from the client. The peer connection choice strategy administrates the pool of connections with other peers. It decides which connections are closed and what new connections should be established. Finally, the request scheduling strategy determines how requests received from remote peers are scheduled for transmission.

We chose this decomposition in order to provide basic implementations of the most important aspects of the BitTorrent protocol. All four provided strategy classes have no inter-dependencies. This allows them to be re-used for the implementation of further BitTorrent-based protocols. Protocols which, as the one presented in [20], alter the part selection strategy but otherwise make no claims about further changes to the protocol logic can hence be built simply by implementing the new part selection strategy and linking it with the other basic implementations in the provided protocol factory. This factory is called by each `BitTorrentClient` class instance upon start-up and creates the needed set of classes.

Our implementation of the decomposition shown above proved to be in no need for inter-strategy communication. As a consequence, we have not yet implemented an inter-strategy communication scheme. However, we recognized that there is some potential for synchronization among the strategies. For example, it can be beneficial for a part selection strategy when certain peers which have interesting parts get unchoked so that these peers, in turn, may also unchoke the initiating client in a "tit-for-tat" manner. For this, the part selection strategy would have to request whoever is in charge of unchoking to unchoke that specific peer. We are hence planning to integrate a broadcast communication scheme among the active strategies with a standardized interface for requests and responses.

4.2.1 Part selection

The original BitTorrent protocol subdivides requests for each piece into smaller-sized block requests to avoid downloading large chunks of data from slow or overloaded peers. Our part selection strategy implementation adheres to this principle and maintains two associative arrays, one for all blocks which have to be requested in order to complete the download as well one for all requests sent out to a specific remote peer. An internal scheduling function is triggered whenever a new client is connected, a new `have` message is received, or if a requested block has been received or has timed out. This scheduling function iterates through all available peers, checks whether it is allowed to send requests to that peer (i.e., the peer is not choking and a maximum threshold of requests per peer is not exceeded) and calls the `GetHighestPriorityBlockForPeer` method which determines which block to request next. When a block to request is found, a request for it is issued via the `Peer` class and a timeout event for the request is inserted into the simulation's global event queue. In the standard implementation, blocks (and hence, full pieces) are requested in sequential order, from lowest to highest. In order to determine available pieces, the strategy listens for connecting and disconnecting remote peers and their sent bitfield messages, as well as for arriving have messages. A derivative class implements the BitTorrent-default rarest-piece-first strategy by overriding the `GetHighestPriorityBlockForPeer` method and also listening to the aforementioned events, keeping track of the peers that announced a certain piece.

4.2.2 Choking/Unchoking

The choking/unchoking of BitTorrent is represented by a strategy which works in accordance with the algorithm definition found in [1]. Periodically, upon client connection/disconnection, and upon the reception of "(un)interested" messages, the strategy calls a method which dissects the set of remote peers into those which "choke", "unchoke" and no choking-related messages are sent to. The decision to unchoke a peer (`GetPeersToUnchoke` method) is based on the perceived performance of a peer according to the download rate, which is calculated automatically by the `Peer` class in a rolling-average fashion based on the total amount of TCP payload data received from the client. Note that this is sufficient because of the usually relatively small overhead of the Peer Wire Protocol compared to the received file data. The top peers are sent "unchoke" messages (if not already unchoked) and peers displaced by the new top ones (`GetPeersToChoke`) are sent "choke" messages. We also implemented BitTorrent's optimistic unchoking scheme by periodically sending an "unchoke" message to a biased-randomly-selected peer and choking the old one, if necessary.

4.2.3 Peer connection choice

Whereas the communication between already-connected BitTorrent clients is relatively easy, BitTorrent's peer discovery mechanism is not since the original protocol involves a HTTP-based tracker system. In order to support seamless operation in emulation scenarios, we had to incorporate a minimalistic HTTP client model into our base peer connection strategy. This model periodically opens a TCP-based HTTP connection with an instance of the tracker model described in section 3.1.3 and, upon the reception of the tracker's answer, adds the addresses of new clients it received to the client's list of known peers. Since the server's answer by protocol standard includes nothing but the addresses of peers, the strategy then randomly establishes connections by instantiating new `Peer` objects until a client-specific connection threshold is reached. The strategy also takes care for any timeouts which may occur during both peer- and tracker connection establishment and also assures that at most one connection between two peers exists. Not implemented is an automatic closing of a connection in cases of Peer Wire protocol errors such as too large requests (see [6]) since some experiments may very well lead to non-standard behavior and in such cases directly closing "off-bound" connections would be counterproductive.

4.2.4 Request scheduling

The request scheduling strategy is intended to provide a counterpart to the part selection strategy on the sending side, which may prioritize requests received from peers. In our basic (stub) implementation, this strategy solely checks whether the requesting peer is unchoked and the requested block is locally available. If so, it directly issues the initiation of the block transfer, otherwise it silently drops the request. We want to note that using this scheme, the available upload bandwidth is equally shared among all peers a client has unchoked. This may result in a degradation of overall performance if too many peers are unchoked. However, we opted against the implementation of a more elaborated traffic scheduling policy since BitTorrent's choking mechanism was essentially created to mitigate problems arising from TCP's congestion control (see [5]). In our basic implementation, it is hence the responsibility of the choking/unchoking strategy to provide sufficient upload speeds to every unchoked peer, either by using proven standard settings such as those found in [1] or by using more advanced techniques taking into account the available upload bandwidth, such as the one presented in [12].

4.3 Communication wrapping

Communication with remote peers, as pointed out before, is carried out by instances of the `Peer` class, one instance per connected peer per client. The `Peer` class capsules a `TcpSocket` object used to establish the remote connection. To transmit data to the remote peer, the `Peer` class provides methods for sending out each available message as specified in [1] as well as for the BitTorrent Extension Protocol specified in [14], with arguments as appropriate for the type of message to send. When such a method is called, the message is created in form of derivatives of ns-3's `Packet` class and appended to an internal queue of the `Peer` object. In case of a "piece" message, i.e., a block of actual file data, the `StorageManager` singleton is called to retrieve the appropriate part of the file to transmit. BitTorrent's "have" and "cancel" messages, which represent changed state information and messages used to prevent unnecessary data transmission, respectively, are prioritized by prepending them. This queue is emptied by a method triggered by the `Socket`'s send callback which also takes into account possible buffer overflows of the socket. Data reception is handled by a small state-machine which empties the TCP socket's reception buffer into a further internal buffer. Since all BitTorrent messages include length-prefixes and are not interleaved, the end of a message reception is easy to determine and messages are only processed after they have been fully received. As a consequence, our model does not support the announcement of a starting or ongoing block transmission, since this would have introduced too many additional notification events with little information gain (note that the SHA-1 hashes in BitTorrent are on piece level, meaning that a correct download of data can only be announced when a full piece was completed). Each processed message is announced through the client's internal event dispatcher; the contents of the received message is supplied to the registered event listeners as function parameters. Our implementation allows one to use real SHA1 hashes for the verification of pieces in order to achieve full compliance with other BitTorrent systems. The default configuration of the model, however, discards the checksum calculation entirely to reduce the overall simulation run-time.

```
[...]
0h0m0s: topology set node count 125
0h0m0s: client 1 set initial bitfield full
0h0m0s: clients 2 till 125 join group
        leechers
0h0m0s: group leechers set initial bitfield
        gaussian mean 0.5 stddev 0.2
        random tail
0h0m0s: client 1 init
from 0h0m15s till 0h10m15s: group leechers
        init
1h0m0s: group leechers set strategy options
        request_size=16384 max_peers=42
[...]
```

Figure 3: Excerpts from a simulation story file.

4.4 Scenario Setup in Stories

To aid the inevitable scripting of swarm behavior, we decided to implement a scenario setup manager which takes a human-readable plain-text file (to which we refer as a scenario's "story") and generates the desired swarm behavior by triggering appropriate actions offered by our client model. For this purpose, our `BitTorrentClient` class offers a number of public member methods which grant access to general client actions and settings (such as joining/leaving the swarm and the client's initial download status represented by its bitfield) as well as a generalized interface for strategy-specific parameters (such as the size of block requests). The `StoryReader` singleton then for each BitTorrent client-related event inserts the corresponding method call with appropriate parameters for each affected client into the simulation's global event queue. Single-client events can be given exact points in time and events affecting multiple clients (which may be joined into client groups) can also be assigned "fuzzy" times, meaning that the events are distributed randomly within a given time range (cf. figure 3 for an example). Moreover, the `StoryReader` can be used

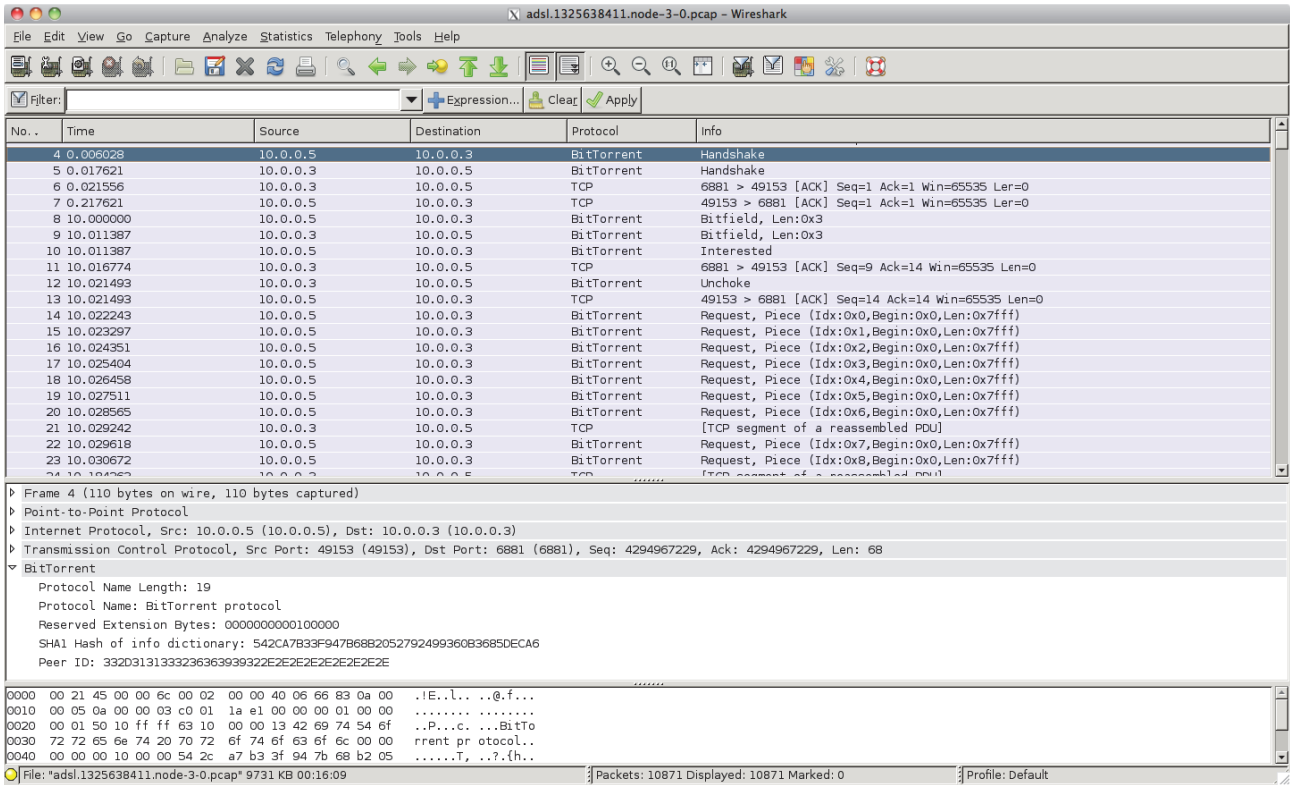


Figure 4: Wireshark correctly dissects a PCAP trace produced using our BitTorrent Model. This demonstrates that the peer wire protocol as implemented by our BitTorrent client model is compatible with real-world BT software.

to set globally-effective settings such as those of the tracker model, the used ".torrent" input file and the random seed (to enable repeatable simulations). Additionally, it can also be used to create nodes in a given network and install BitTorrent clients on them when interfaced with a specialized version of a topology reader. For the latter, our model currently supports working on BRITE topologies using a simplistic self-written topology reader.

Note that our whole BitTorrent model may also be scripted using ns-3's common C++ or Python scripts and that the `StoryReader` is intended as a mere convenience interface to the simulation model. During our initial tests, however, it showed that using our story setup utility greatly speeded up simulation setup as especially node grouping and -retrieval and scheduling of method calls was greatly simplified, leading to far better readability of our "story" scripts than those written in one of ns-3's "native" languages.

5. PRELIMINARY EVALUATION

The preliminary evaluation in this paper investigates two aspects of our BitTorrent model, namely the validity of the peer wire protocol implementation and the performance of the client model.

5.1 Peer-Wire Protocol Validity

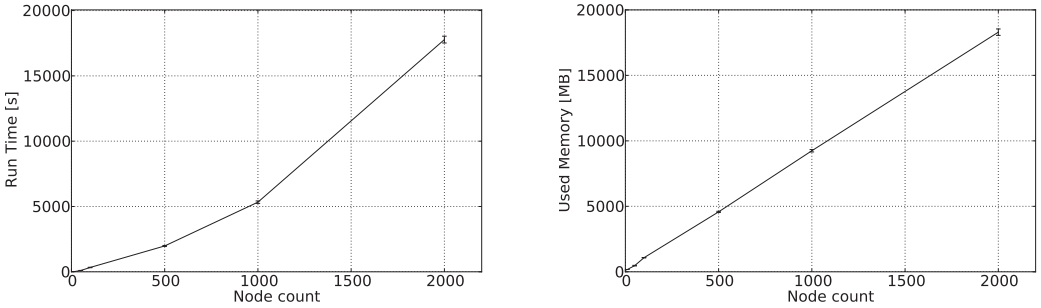
In order to model the peer wire protocol in a realistic fashion, it is important to accurately reproduce its protocol semantics and to correctly recreate the protocol messages.

We validated the protocol semantics by adding a large amount of logging statements to all the components of our model. We then compared the logging output and the logical sequence of the messages with application traces obtained from a real-world BitTorrent client (Unworkable [2]) and iteratively adjusted the behavior of the BitTorrent client model. In the present stage of the model the handshake and all standard messages are fully compliant to the BitTorrent specification BitTorrent-spec-1.0.

We also investigated if the messages passed among the simulated clients resemble the message formats used by real-world BitTorrent clients. For this purpose, we first investigated the message compatibility by obtaining a PCAP trace from a BitTorrent simulation and by opening the trace in the Wireshark [4] network protocol analyzer (see Figure 4). Wireshark correctly dissects the packets and the information displayed in the protocol analyzer is consistent with the behavior specified in the simulation, which uses a pure sequential retrieval of pieces in the depicted example. Secondly, we set-up a simple emulation scenario containing one simulated tracker, one simulated seeder and one Vuze [3] BitTorrent client. Vuze was able to correctly download the file from the BitTorrent simulation, with the download speed matching the link capacity specified in the story file.

5.2 Simulation Performance

We now investigate the overall simulation performance of our BitTorrent client model by looking at two performance metrics, *memory consumption (measured in Megabytes)* and *simulation run-time (seconds)*. For this purpose, we used a star topology in which a varying number of BitTorrent clients is connected to one central router using point-to-



(a) Simulation Run-Time vs. Node Count (b) Memory Use vs. Node Count

Figure 5: We evaluated the performance of our BitTorrent client model by investigating the simulation run-time and the memory use for different node counts. With our simulation model, it is possible to simulate also larger BitTorrent swarms if sufficient memory resources are available.

point links. The link capacity for each client was set to 448 kbps on the uplink and 2048 kbps on the download, resembling a typical ADSL line. For all node counts, we use a seeder/leecher ratio of 2/3 in our simulations as initial swarm configuration. All simulated leechers start requesting the data at the same time, yielding to a flash-crowd behavior. The payload data size exchanged among the nodes was 10 MB, the piece size was configured to 128kb. All runs were carried out on a Intel Xeon computer with 24 cores at 3.33 GHz each. No parallelization was used to speed up the execution. All presented results are averages over three simulation runs. We observed very little variations in the measured simulation run-time and the memory required.

The outcome of our measurements is depicted in Figure 5. The run-time of the simulation rapidly grows for larger node counts. However, the simulation run-time for the largest experiment with 2000 nodes was still shorter than five hours, indicating that even experiments with mid-size swarms can be carried out with this simulation model

The larger issue are the memory requirements of our simulation model. The memory required for the BitTorrent model grows linearly with the nodes in the simulation, resulting in quite high RAM requirements. The experiment with 2000 nodes used over 18 GB of RAM. This shows that packet-level simulations of mid-size BitTorrent swarms are possible on machines available today if sufficient RAM capacities are available. We are currently investigating code optimizations to lower the RAM requirements.

5.3 Download Completion Time

In addition to the simulation's performance we also briefly investigated the downloading performance of our BitTorrent client model. Figure 6 visualizes the download completion time for all 1333 leechers in the 2000 node experiment. The vast majority of the clients completes the download between 220 and 280 seconds, demonstrating a coherent model behavior. The theoretical minimum to transfer 10MB over a 448kbitps link would be 178 seconds and 39.06s for a 2048kbit/s link. We explain these large differences of downloading performance by the protocol overhead of BitTorrent, the flash-crowd nature of the scenario and the fact that our rarest-first strategy is yet unoptimized. However, these results also indicate that a further validation and adaptation of the model is needed before it can be used for real BitTorrent experiments.

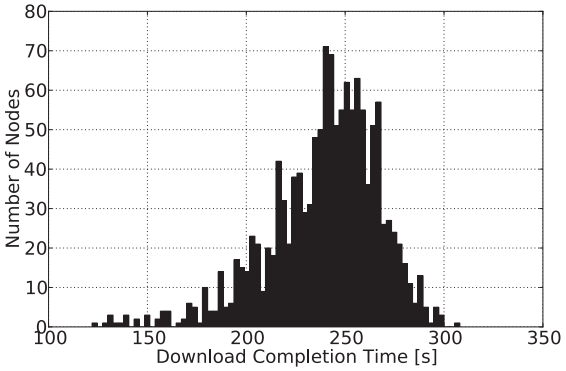


Figure 6: Distribution of the download completion time of 1333 leechers in a simulated BitTorrent swarm. Most of the clients complete the download within 220 and 270 seconds.

6. LIMITATIONS

Our model currently only implements the basic ingredients of the BitTorrent protocol. Except for the Extension Protocol (see section 4.3), none of the more than 20 proposed extensions, such as DHT-based operation or UDP-based communication with the tracker, are currently supported. However, note that only few real-world clients support a large subset of these proposals. A further limitation of the client model is the currently missing support for traffic shaping on a per-peer basis. Some BitTorrent clients like BitTyrant [16] regulate the outgoing traffic to some of their peers in order to improve their own downloading performance, while our implementation currently lacks a strategy to impose these restrictions, preventing the simulation-based recreation of the behavior of such clients and traffic-shaping-based strategies.

A general shortcoming is the limited validation of the model behavior we conducted so far. While our simulation model correctly recreates the behavior of the BitTorrent peer wire protocol and the communication with the tracker, we still need to validate how much the download performance of our client model resembles that of real-word BitTorrent systems.

7. RELATED WORK

To our knowledge there is no BitTorrent model for the ns-3 network simulator so far. The only P2P models for ns-3 we are aware of target the Chord DHT [18] and Pastry [17]. However, different BitTorrent models already exist for other network simulators. GPS [22] is a specialized discrete event-based simulator for BitTorrent. While it uses an exact model for the BT protocol, it abstains from modeling data payload and uses a flow-based approach to recreate the behavior of TCP streams. This makes the model very efficient, but hinders its use e.g. for emulation purposes. There are also BitTorrent models for specialized P2P simulators like PeerSim [13] or even a distinct BitTorrent simulator by Microsoft [11]. Similar to GPS, all these models and simulators mostly model TCP/IP only at an abstract level in order to improve the efficiency of the simulation. Eger at al [7] compare the accuracy of flow-based BitTorrent models with outcome of a packet-based BitTorrent model for ns-2 [15]. They conclude that although flow-based and packet-based models yield to similar results, the latter are required to study cross-traffic interaction. Another packet-based BitTorrent model has been proposed for OMNeT++ [19] by Katsaros et al [9]. While the model seems to well recreate the behavior of BitTorrent, the use of the INET framework with its proprietary message formats hinders the use for emulation studies. In contrast to all the BitTorrent models proposed for other simulators, we aim at reproducing not only the BitTorrent protocol mechanics, but also the packet formats in a way that we achieve message compatibility with real BT software. We are aware that this will limit the scalability of the model to a certain degree, but will enable simulation-driven emulation studies with BitTorrent.

8. CONCLUSION AND FUTURE WORK

In this paper we have presented the design and the implementation of a BitTorrent model for ns-3. To our knowledge, this is the first BitTorrent model for this simulator. It uses the same message formats as used by real-world BitTorrent systems and allows for the simulation of BitTorrent swarms with hundreds to thousands of nodes. We are currently performing further validations of the BitTorrent client model and investigate both experiments in the context of BitTorrent-based Video-on-Demand systems and an integration with SliceTime [21] for large-scale BitTorrent emulation studies. The source code of the model is available at `www.comsys.rwth-aachen.de/research/projects/vodsim/`.

Acknowledgements

The authors thank their anonymous reviewers, Alexander Hocks, Suraj Prabhakaran and the members of the QuaP2P research group for their valuable feedback. This work was partially funded by DFG grant 733 and the UMIC excellence cluster.

References

[1] Bittorrent protocol specification v1.0. `http://wiki.theory.org/BitTorrentSpecification` accessed 11/2011.

[2] The unworkable BitTorrent client. `http://code.google.com/p/unworkable/` (accessed 11/2012).

[3] Vuze. `http://azureus.sourceforge.net/` (accessed 02/2012).

[4] Wireshark network protocol analyzer. `http://www.wireshark.org` (accessed 02/2012).

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

[6] B. Cohen. The BitTorrent Protocol Specification, 01 2008.

[7] K. Eger, T. Hoßfeld, A. Binzenhöfer, and G. Kunzmann. Efficient simulation of large-scale p2p networks: packet-level vs. flow-level simulations. In *UPGRADE-CN*, pages 9–16. ACM, 2007.

[8] J. Hoffman. BEP 16 - Superseeding (draft). `http://bittorrent.org/beps/bep_0016.html`, 02 2008.

[9] K. V. Katsaros, V. P. Kemerlis, C. Stais, and G. Xylomenos. A bittorrent module for the OMNeT++ simulator. In *MASCOTS*, pages 1–10. IEEE, 2009.

[10] A. Loewenstern. BEP 05 - DHT Protocol (draft). `http://bittorrent.org/beps/bep_0005.html`, 02 2008.

[11] Microsoft Research. Microsoft BitTorrent simulator. available online at `http://research.microsoft.com/apps/dp/dl/downloads.aspx` (accessed 11/2011), 2005.

[12] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips. Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems. In *Multimedia Computing and Networking 2008*, volume 6818. SPIE Vol. 6818, Jan. 2008.

[13] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, 09 2009.

[14] A. Norberg, L. Strigeus, and G. Hazel. BitTorrent Extension Protocol, 01 2008.

[15] The network simulator ns-2. `http://www.isi.edu/nsnam/ns/` (accessed 11/2011).

[16] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in bit torrent. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, Cambridge, MA, USA, 2007.

[17] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, volume 2218 of *LNCS*, pages 329–350, Heidelberg, Germany, Nov. 2001. Springer-Verlag.

[18] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM 2001*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, Aug. 27–31 2001. ACM Press.

[19] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *SimuTools*, page 60. ICST, 2008.

[20] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BitoS: Enhancing bittorrent for supporting streaming applications. In *INFOCOM*. IEEE, 2006.

[21] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. Slicetime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, Boston, MA, 2011.

[22] W. Yang and N. Abu-Ghazaleh. Gps: A general peer-to-peer simulator and its use for modeling bittorrent. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:425–434, 2005.