CS 259 Final Project Report: BitTorrent

Nathan Marz and Raylene Yung

March 20, 2008

1 Overview of BitTorrent Protocol

In BitTorrent, the data to be shared is divided into many equal-sized portions called *pieces*. Each *piece* is further sub-divided into equal-sized sub-pieces called *blocks*. All clients interested in sharing this data are grouped into a *swarm*, each of which is managed by a central entity called the *tracker*. Each client independently finds a file, called a *torrent*, that contains the location of the tracker along with a hash of each *piece*. Clients keep each other updated on the status of their download. Clients download *blocks* from other (randomly chosen) clients who claim they have the corresponding data. Accordingly, clients also send data that they have previously downloaded to other clients. Once a client receives all the *blocks* for a given *piece*, he can verify the hash of that *piece* against the provided hash in the *torrent*. Thus once a client has downloaded and verified all pieces, he can be confident that he has the complete data.

As specified in the RFC (http://jonas.nitro.dk/bittorrent/bittorrent-rfc.html), BitTorrent can be described in terms of two sub-protocols: one which describes interactions between the tracker and all clients, and one which describes all client-to-client interactions.

1.1 THP: Tracker HTTP Protocol

THP serves two purposes. First it defines how an agent connects to the tracker in order to register with the swarm, as well as receive a list of other members of the swarm. Secondly, this protocol describes how an agent periodically reports its progress to the tracker.

We note that denial of service attacks against the tracker are already well-known. Also, the BitTorrent protocol is evolving to allow for multiple trackers per swarm, reducing the effectiveness of these attacks. Thus, for the purposes of this project, we chose to abstract away the purpose of the tracker, and concentrated on the second sub-protocol (PWP) instead.

1.2 PWP: Peer Wire Protocol

Once agents are connected to a given swarm, PWP describes all interactions between agents. Essentially, this sub-protocol defines the necessary functions of the protocol.

It describes the following actions:

- agents notify one another of missing and completed pieces
- agents contact one another to request data
- agents connect to one another to send data

2 Security Properties

The attacker we consider has as much power as any other agent in the network. Our attacker does not have control of the network. As will be described later, BitTorrent lacks authentication and it is possible to forge any message from any agent at any time. We wanted to study how strong of an attack could be done even with this limited adversary. There are two important security properties to consider:

- Integrity: An attacker should not be able to force a client to accept data not used to create the torrent file. This property is trivially fulfilled since before a client accepts downloaded data as correct, he verifies the hash in the *torrent*. In other words, the data is as correct as the strength of the hash.
- Resistance to DOS: An attacker should not be able to cause the download rate for any client to fall below a certain low threshhold, without doing too much work himself. We chose to study this property.

3 Modeling of Protocol

As mentioned in the overview of the protocl, BitTorrent uses randomization to decide which clients a given client chooses to download from, as well as which *block* to request. Thus, we chose to use PRISM as our model checker because it was the only one that can model randomness. We chose to model all normal agents deterministically with random choices, while the attacker behaves non-deterministically.

3.1 Problems with PRISM

Although PRISM was the most suitable model checker available, there are inherent difficulies with using PRISM to model complex protocols.

3.1.1 Symmetry and Scale

In PRISM, there are no complex data structures. In other words, every agent must be modeled as its own set of variables and transitions. This is both time-consuming and unintuitive to design by hand, so we chose to use PHP to dynamically generate PRISM code. Our final PHP script takes the parameters *(number of pieces)*, *(number of blocks per piece)*, and *(number of agents)*. All of our resulting models assumed one *seeder* and one *attacker*.

3.1.2 Complex Logical Structures

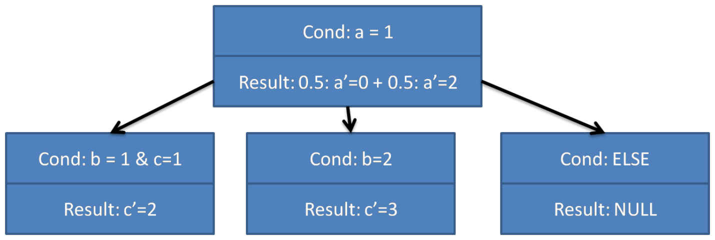
In order to model interactions between agents, nested logical if-else structures are used in the BitTorrent protocol. For example, we would like to model:

```
If received block "l" for piece "p"
    If all blocks downloaded for "p"
        If "p" is verified (hash matches torrent-hash)
            set status of "p" to downloaded/verified
        Else clear statuses for all blocks in "p"
```

However, PRISM only allows commands of the following form:

```
[condition]->[probability distribution of results]
```

In order to solve this problem, we created a data structure in PHP called *TieredTransition*. A *Tiered-Transition* has the following structure:



Each path from the root to a leaf results in one PRISM command, where the resulting condition is the conjunction of all conditions along the path, and the resulting probability distribution is computed from combining conditional probabilities along the path. In the above example, our script generates the following PRISM code:

```
[] a=1 & b=1 & c=1 -> 0.5: (a=0) & (c=2) + 0.5: (a=2) & (c=2);
[] a=1 & b=2 -> 0.5: (a=0) & (c=3) + 0.5: (a=2) & (c=3);
[] a=1 & !(b=1 & c=1) & !(b=2) -> 0.5: (a=0) + 0.5: (a=2);
```

An advantage of this data structure is that we can easily define the *ELSE* construct by taking the conjunction of the negations of a node's siblings.

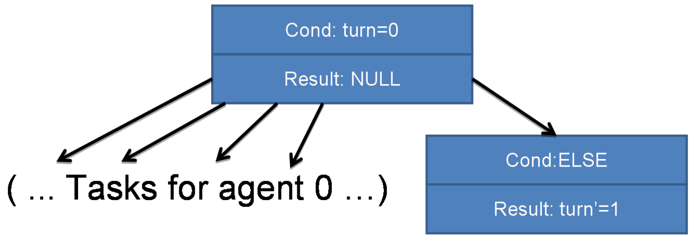
3.1.3 Nondeterminism

In BitTorrent, agents operate independently. However, if we were to model agents independently in PRISM, they would act nondeterministically. Subsequently, possible attacks found in PRISM would include nondeterministic decisions made by both the attacker as well as all other agents. Additionally, if agents are modeled independently, it is possible for agents to be completely inactive if PRISM chooses never to follow their transitions. In order to solve these problems, we used two techniques: turn based modeling and sequential task ordering.

(a) Turn Based Model

In our turn based model, agents act in a round-robin manner. We added a global variable indicating the id of the currently active agent, and incremented it accordingly. By adding the value of this turn counter variable to each transition, we ensure that a given agent's transitions are only followed during their turn. Additionally, every agent gets a chance to act.

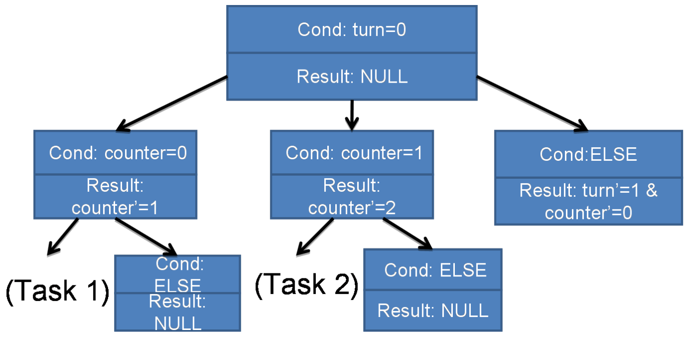
Here is a sample turn-based transition model for $Agent0$.



The *tasks* referred to above are actions such as responding to a request for a block, sending requests, notifying other agents that a piece was just downloaded, etc.

(b) Sequential Task Ordering

As mentioned above, an agent must act deterministically on their turn. However, many of their tasks may be independent, such as sending a block versus receiving a block. Additionally, there are conditions to performing each task, so not all tasks need be performed every turn. We achieve determinism within each turn by imposing an ordering on the tasks. Like the turn-based counter, this is implemented via a global task-counting variable. Each task has an associated id. When this variable is set to a particular task's id, the condition required to perform the task is checked. Regardless of whether the condition holds (and the task subsequently performed), the variable is incremented. A *TieredTransition* exemplifying this technique is shown here:



3.2 Resulting Model

Unfortunately, after implementing the above techniques, our PRISM models were extremely large. Even after implementing several optimizations, the largest feasible model we could construct consisted of 2 *agents*, 2 *pieces*, 1 *block per piece*, 1 *seeder*, and 1 *attacker*. Anything larger caused PRISM to run out of memory. These models are too small to emulate any realistic scenario.

3.2.1 Optimizations

As mentioned above, in order to reduce the size of our model we implemented several optimizations:

- the *seeder* does not request blocks from anyone else
- the *seeder* does not send state updates to anyone else
- the agents do not send blocks to the *seeder*

3.3 Attacker Design

Designing a model for an attacker in a randomized protocol is particularly difficult. Even though an attacker can forge any message at any time, allowing this amount of nondeterminism would be unrealistic. For example, suppose *Agent*1 makes a request for a block from *Agent*2 and is now expecting a response. An attacker has an opportunity to forge bad data that appears to come from *Agent*2 but must do so before *Agent*2 responds directly to *Agent*1. If the attacker was allowed this much nondeterminism, the model checker would assume the attacker always preempts other agents, as in this example. This is because the model checker chooses nondeterministic choices so as to maximize the probability of a successful attack. However, in practice, this amount of nondeterminism is unrealistic because an attacker *lacks this knowledge*. In our design, the attacker has nondeterminism in the attack, but for parts where the attacker requires a certain amount of knowledge, he must use randomness to *guess* any knowledge he was unable to learn.

One could also try using probabalistic models so that the attacker can make better guesses and have better attacks. However, the amount of effort this entails makes the idea of using model checking to analyze BitTorrent less appealing. Instead of defining a few simple rules for the scope of possible attacks, as can be done with deterministic protocols in Murphi, constructing such complex probabilistic models is equivalent to determining the attacks by hand. Thus, modeling the attacker is yet another weakness of using a probablistic model to model a complex randomized protocol.

4 Security Analysis

Through the process of constructing the model, we discovered some intuitions for how denial of service attacks would work on the PWP protocol. Although, as mentioned above, automated model checking was infeasible, we were able to use Rational Reconstruction techniques to discover some interesting protocol weaknesses.

4.1 Spoofing

BitTorrent, as defined in the RFC, does not require any authentication of PWP messages. In other words, an attacker can forge any message under the guise of any other agent. Due to this, an attacker can easily change agents' *perceptions* of other agents. For example, an attacker could make $Agent\,1$ think $Agent\,2$ has some piece p even though this isn't true. Accordingly, $Agent\,1$ might waste resources and time requesting various blocks from p from $Agent\,2$. Furthermore, after repeated unfilled requests, $Agent\,1$ might mistakenly believe $Agent\,2$ to be an unreliable source of data, and stop sharing data with them. In the worst case, in this example the connection $Agent\,1$ and $Agent\,2$ could be completely severed.

Interestingly, through Rational Reconstruction we discovered that the handshaking protocol of BitTorrent provides some protection against a spoofing attack. During a handshake, the two agents involved will exchange bit-arrays encoding the completed and missing pieces for each party. Once a handshake is completed, the two agents are considered to be *connected* and eligible to share data. The BitTorrent RFC suggests that given some unusual behavior by either party, the connection is severed until a new handshake is initiated and completed. However, the RFC does not clearly define the conditions upon which a connection should be re-established. This is one area in which the RFC can be greatly improved. We feel that *anytime* an unexpected message is received, the connection should be severed.

In the example above, once $Agent\,2$ receives a request for a piece it does not have, it should sever its connection with $Agent\,1$ until another handshake is completed.

4.2 Fake Piece Messages

As described before, to download a piece, an agent downloads each block for the piece separately and possibly from different agents. If the piece fails to verify once all blocks are downloaded, the agent must re-download every block because he doesn't know which block was bad. In a real world torrent, there can be more than 200 blocks per piece. So for every bad block an attacker succeeds in getting an agent to accept, the agent has to do *200*$\times$ more work.

The RFC is not clear in what an agent should do if he gets a block from someone he didn't request it from. Clearly, based on the analysis above, an agent should not accept any blocks that were not specifically requested. This would make it much more difficult for an attacker to succeed in getting an agent to download a bad block. To be successful, an attacker would have to guess both which blocks a given agent is missing, along with who the agent requested them from, if anyone. However, because the ratio of work done by the agent to the attacker is so high, 200:1, even if it took an attacker 30-40 guesses per success, this would still be an effective attack.

There are some additional messages that could be added to the protocol to make this sort of attack impossible. Consider the following modification to the protocol:

- Agent A requests block B from agent C, and in the message adds an additional random nonce R
- Agent C sends (C,B,R) to A
- Agent A accepts B if it comes from the correct agent and the nonce is the same

Since our attacker model does not have control of the network, and thus cannot intercept messages, this modification makes it impossible for an attacker to forge a block message.

Given this modified version of the protocol, an attacker can still cause an agent to accept bad blocks if the agent sends requests directly to the attacker. However, if an agent downloads all blocks for a piece from just a few agents, he should be able to quickly learn which agents are "bad" and stop making requests from them.

5 Conclusion

Our results from the analysis of BitTorrent were threefold. First, we discovered many interesting higher-level techniques for constructing probabilistic models and dealing with nondeterminism. We believe these techiques can apply to any complex system modeled in PRISM. Secondly, we found limitations of PRISM itself. We realized it was not suitable for the analysis of BitTorrent because of the size of the state and variable space. Alternatively, PRISM is more suitable for a protocol with a small fixed number of messages. Finally, we found several security weaknesses in BitTorrent and proposed modifications to the protocol which solved these issues.