Faithfulness in Internet Algorithms

Jeffrey Shneidman, David C. Parkes
Division of Engineering and Applied Science
Harvard University
{jeffsh, parkes}@eecs.harvard.edu

Laurent Massoulié
Microsoft Research Ltd.
Cambridge, UK
{lmassoul@microsoft.com}

ABSTRACT

Proving or disproving faithfulness (a property describing robustness to rational manipulation in action as well as information revelation) is an appealing goal when reasoning about distributed systems containing rational participants. Recent work formalizes the notion of faithfulness and its foundation properties, and presents a general proof technique in the course of proving the *ex post Nash* faithfulness of a theoretical routing problem [11].

In this paper, we use a less formal approach and take some first steps in faithfulness analysis for existing algorithms running on the Internet. To this end, we consider the expected faithfulness of BitTorrent, a popular file download system, and show how manual backtracing (similar to the the ideas behind program slicing [22]) can be used to find rational manipulation problems. Although this primitive technique has serious drawbacks, it can be useful in disproving faithfulness.

Building provably faithful Internet protocols and their corresponding specifications can be quite difficult depending on the system knowledge assumptions and problem complexity. We present some of the open problems that are associated with these challenges.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; J.4 [**Computer Applications**]: Social and Behavioral Sciences—*Economics.*

General Terms

Algorithms, Design, Economics.

Keywords

Backtracing, Computational Mechanism Design, Distributed Algorithmic Mechanism Design, Computational Failure Models, Faithfulness, Program Slicing, Rational Failure, Rational Manipulation

1. INTRODUCTION TO FAITHFULNESS

The economics subfield of *mechanism design* (MD) studies how to build systems that exhibit good behavior *in equilibrium*, when self-interested participants pursue self-interested strategies [17]. Computer scientists have recognized that ideas from MD may be applicable to algorithm design when multiple self-interested and competing nodes are interested in an algorithm's outcome. Extensions to MD may also help distributed algorithm design when the work of the algorithm calculation can be pushed back onto the very nodes that are interested in the calculation.

Traditional MD assumes that the self-interested strategies consist entirely of information revelation actions; i.e., a node may strategize only about their declared inputs to an algorithm created by the system designer. In a network setting, this limitation is not valid since one cannot assume an obedient networking and mechanism infrastructure when rational players control the message paths and mechanism computation. In this setting, a node can strategize about all actions related to its algorithm involvement, rationally manipulating part of an algorithm in order to try to selfishly better its outcome.

In concluding their seminal work on *algorithmic mechanism design* (AMD) [14, 15], Nisan and Ronen refer to this as the "set of problems" that come in implementing a mechanism in a network [16]. In their work on *distributed algorithmic mechanism design* (DAMD) [7, 8], Feigenbaum et al. discuss these problems as "the need to reconcile the strategic model with the computational model."

To address these issues, Shneidman and Parkes present a treatment of rational manipulation in distributed systems, and provide a framework and a way of reasoning about faithful behavior in mechanisms [11]. That work formalizes the notion of *faithfulness* and its foundation properties of *incentive-, communication-, and algorithm-compatibility* that when proven together guarantee that participating nodes are faithful to a suggested specification. In contrast to that paper's formal approach, which is used to prove faithfulness of a theoretical interdomain routing problem, our paper presents a technique that can be used to *disprove* faithfulness and locate rational manipulation points in specifications. To our knowledge, this type of analysis hasn't been widely attempted by systems designers to find parts of an algorithm or protocol that are susceptible to rational manipulation. Although the technique is imperfect, it was useful in finding several weaknesses in the BitTorrent algorithm, the first algorithm that we have evaluated.

For the purposes of this paper, we can use an informal definition of *faithfulness*, leaving the formal discussion to earlier work [11].

An interesting aspect of protocols and algorithms running on the Internet is that in many cases, there is a default software program or official protocol specification that is supplied to clients. A system designer can use this to convey a default or suggested behavior that it wishes a node to follow. This allows the mechanism designer to suggest an equilibrium *strategy*, which is a plan or decision rule that specifies which *actions* this node will take for all possible states of the world.

Definition 1. *A system is* ***faithful*** *with respect to some knowledge assumption if a rational (selfish, utility-maximizing) node follows an algorithm that always has the same externally visible effects as following the suggested strategy, for all states of the world.*

This definition states that given some assumptions about the types of nodes in a system (the economics *equilibrium concept*), a node acting in its own self interest will follow the default strategy (or an equivalent that is indistinguishable to an outsider) in a faithful system. The node follows the default strategy because it is a utility-maximizing strategy for this node, and there is no other strategy that yields a higher utility for this node.

When we say that the system is faithful, we mean that if a specification is written to be faithful when implemented correctly, then a rational node will choose to run a correct implementation. An old saying of software development is that "the source code is the ultimate specification." One interpretation of this adage is that in the absence of a well-defined specification, one needs to refer to the source code to understand a system's behavior. In this case, there is no distinction between a specification and the implementation.

2. SOFTWARE AS A STRATEGY

The software executed by a particular node can be seen as a representation of that node's strategy. Software starts in an initial state. The software receives inputs that with its own information determine the perceived state of the world. The software then moves to a new state, perhaps emitting an *external action*. (A node that takes no external action, perhaps because of a strategic delay, can be said to take the *empty* action.) When economics discusses strategies and actions, all of the internal state transitions and unobservable *internal actions* that occur within software are wrapped together as part of a strategy that generates an *external action* observable to outside nodes. An auction is a good example: the auctioneer may ask a node to bid for an item. The node considers the state of the world, the input of the ask, and after a moment responds to the auctioneer. This process can be described as this node's strategy for a particular state.

With this view of software, each machine code instruction in a program can be thought of as representing a little bit of one or more components of a node's strategy. Each function call has the potential to emit an external action, and/or evaluate and change the perceived state of the world. At the most abstract, a node can perform *any* strategy and *any* action, though a mechanism designer would restrict the actions to a *valid set* that can be expressed. For instance, a designer might impose the restriction that all actions must be framed in valid Ethernet packets, via a network jack. While a node could manipulate the signals on the jack directly, only the subset of all actions that can be expressed as Ethernet packets will be allowed by the physical transport. More commonly the strategy space is restricted to those strategies that can be expressed in a finite multiset of possible executions of machine instructions. A node is often willing to give up some strategy expression in favor of ease of interaction by using obedient *proxies* with restricted languages such as the CPU, operating system, etc. General purpose machines allow a node a sufficient vocabulary to express most relevant strategies.

Software, provided by the system designer, can further restrict the strategy space. This restriction can limit the strategic choice to options that result in outcomes far from a node's ideal outcome. For example, an auction package may restrict a node's valuation expression to one of five numbers, or may devote resources on the node machine to processing the bids of other players in a distributed mechanism. As mentioned earlier, the software can also provide a *default* or *suggested strategy* from the restricted strategy space. While there is a cost to modifying system-provided software, if the utility gained from such a modification is greater than the cost of modification, the rational node will expend this cost. In system terms, this might mean the building of a new software client.

From the mechanism design standpoint, one cares when these modifications come at some cost to other participants, or at some loss of mechanism functionality. There are documented instances of rational manipulation in Internet algorithms that negatively affect other users of the system [10].

For this reason, one should be interested in the *faithfulness* properties of a system, and in the provided client software. Informally, a system is robust to rational manipulation, and therefore faithful, if the designer is able to give an node a piece of source code, a set of suggested run-time options, and state, "Feel free to modify this client as you see fit. One can prove that the best strategy you can follow is the strategy that I have given you."

3. FAITHFULNESS CAVEATS

Systems are often built as a stack of building block components. For example, an operating system provides basic communication primitives, such as sockets. On top of these primitives are other components, eventually including transport protocols like TCP. On top of these components sit applications, such as the BitTorrent system considered in this paper.

When making a faithfulness claim, one should consider strategy manipulations at all parts of the system, and/or declare the level of assumed obedience. For instance, the designer might assert that modifying the firmware running on a network card is too costly for all anticipated rational players. This assertion could also be made for aspects of the operating system, arguably more so if the system is closed source.

This assertion can even be made about the application software itself, especially if the "knobs" to change a default strategy are hard to access. Changing a command line parameter is easier than modifying a configuration file, which is easier than changing compiled opcodes in an executable.

If the expected utility gain from a manipulation is lower than its cost, then a rational person won't make the manipulation. It's important to note that utility isn't strictly equal to, say, "increased performance" in a peer to peer sharing application. It can also include the good feelings of community-endowed prestige or respect for interesting or useful work.

It is possible that one person's work can lower the cost of manipulation of others; one user can modify and spread a client to expose "hidden" details as configurable options. So although the average person might not choose to pay a modification cost given the size of the expected gain, that person may be happy to adopt someone else's work.

With these thoughts, we can now turn our attention to a particular Internet system.

4. FAITHFULNESS IN BITTORRENT

BitTorrent [4] is a distributed file downloading system where peers (a.k.a. nodes) can download a file from each other in addition to the original source. The basic file distribution method is that a server divides a file into pieces, and provides a piece of that file to some peer, which in turn can serve that piece to other peers. Peers build up a file by filling in missing pieces with pieces held by other peers or the original file publisher. This parallel access scheme [18] is useful when a highly anticipated large file is released, such as the latest Linux kernel. Rather than serving a large file in its entirety to a small set of users that provide no additional help to the system, these schemes work by spreading the serving responsibilities across the interested peers. This helps avoid inundating a single file publisher with download requests.

BitTorrent incents users to participate in this distributed serving scheme by linking the receipt of pieces to the serving of other pieces. The idea is that a node should send file data to other peers because this will have the effect of speeding up its own downloads. The incentives used to achieve this are described in the next section.

BitTorrent, representing a class of parallel downloading systems [9] is an interesting system to investigate for four reasons:

First, the system was designed with rational behavior in mind. The system is self-described as using "tit-for-tat as a method of seeking Pareto efficiency" [4]. In other words, there is some idea of incenting good strategic behavior from a rational user. It is useful to understand the strengths and weaknesses of the incentive scheme.

Second, the system designer has provided a suggested strategy; there is a default BitTorrent client that is released in source code and executable form. The client has a default behavior, and there are few switches that can be controlled by the user. The fact that the client source code is available and reasonably small (about 5700 lines of Python script) means that the cost for changing the strategy space is potentially small.

Third, the BitTorrent system is simple enough that the designer can anticipate and model several classes of rational users. These might include nodes who prefer to download a file without using their upload bandwidth, those who prefer to get the file as fast as possible, those who prefer to prevent others from receiving the file, etc. One can study how BitTorrent fares in the face of nodes from each type of rational behavior. Other classes of rational behavior are possible, as will be discussed shortly.

Fourth, the BitTorrent system is popular. The package and its derivatives have been adopted for file distribution by several commercial entities [2, 12] and numerous individuals. It is worth investigating the strengths and weaknesses of real protocols.

4.1 BitTorrent Specifics

BitTorrent is under active development. The description of the system and the analysis in this paper was done on the official BitTorrent client, version 3.2.1b. As of this writing, the current version of BitTorrent is 3.4.2, and there are numerous alternate clients available. Cohen provides a short overview of the system, focusing on the role of incentives [4]. Relevant pseudo-code is given in Figure 1.

BitTorrent works in the following fashion: A system-trusted obedient *tracker* node maintains a list of peers that are active with respect to a particular file. *Complete* peers that hold the entire file, and *incomplete* peers that are likely in the process of downloading the file are tracked. On entering the system, a peer announces itself to the tracker and requests a random subset from the list of peers. A peer then attempts a bidirectional TCP connection to members of the subset.

Of a peer's many established TCP connections, a small subset k is internally marked by the peer as being *unchoked*. A unchoked connection is one where the other endpoint can request a piece of the file, and this peer will fulfill that request. While this doesn't affect the unchoking decision, peers send special messages to each other to update their claim about which pieces they possess. This information allows a peer to request a valid piece. A peer is always capable of receiving a file piece from any connection, regardless of the internally marked choke state. This fact is the basis for optimistic unchoking, described below.

Any active peer that is complete is *altruistic* and will send pieces of a file to other peers in the system, preferring (unchoking) those peers that can download pieces the fastest. An incomplete peer, on the other hand, will selfishly unchoke peers that are currently providing pieces at the highest throughput.

Both complete and incomplete peers will *optimistically unchoke* a new peer at some interval in an attempt to find a better trading partner with higher throughput. On a separate interval, the k fastest partners are kept in the unchoked state, and the remaining peers are choked.

While the system doesn't require more than one altruistic peer to exist, the default behavior of the client is to remain running after the peer becomes complete, becoming an additional altruistic peer. Since the ideal BitTorrent file is large (hundreds of megabytes or larger) and the transfer times are long, people often leave their client running in the background. These additional altruistic peers help greatly in speeding up downloads for the remaining peers.

4.2 BitTorrent's Default Strategy

The normal method of running the client doesn't let the user change any of the BitTorrent settings. The client is designed to run automatically when a user clicks on a special web link. While this is restrictive, Cohen has argued that the simplicity of the BitTorrent interface has been a large measure in the program's success [4].

Default client behavior follows the pseudo-code given in Figure 1. The choice of constants has been picked experimentally by the BitTorrent authors to yield "good" performance.

```
/* Pseudo-Code of BitTorrent Algorithm    *
 * Based on Python source v 3.2.1b        */

// Get list of peers from the tracker.
// Called every N seconds and on startup.
get_more_peers() {
  // get list of peers from tracker
  // send start_connection(me) to subset of list
}

// Add a new peer to my connection list
// Called on receipt of start_connection msg
on_start_connection(peer p) {
  // add p into the middle of unchoke_next list
}

// Decide to whom I should send pieces.
// Called every M seconds
_rechoke() {
  if node_is_complete() { // i.e. has entire file
    // unchoke k fastest receivers
  } else {
    // unchoke k fastest senders (reward others)
  }
  // choke remaining peers and place them at end
  // of unchoke_next list.
}

// Optimistically unchoke a peer
// Called every Q seconds
optimistic_unchoke() {
  // pop peer off unchoke_next list and unchoke
}

// Announce that I have this piece to my peers
// Called when I receive a piece
on_piece_receipt(piece x) {
  // send have_piece(me, x) to my connected peers
}

// Request a piece when I hear annoucement
// Called when I receive a have_piece(x)
on_have_piece(peer p, piece x) {
  if (do_i_have(x) == false)
    send request_for_piece(p, x)
}

// Send a piece when requested by unchoked peer
// Called when I receive a request_for_piece
on_request_for_piece(peer p, piece x) {
  if is_currently_unchoked(p) {
    // send piece x to p
  }
}
```

Figure 1: Pseudo-code of the BitTorrent 3.2.1b algorithm, focusing on the incentives implementation. A BitTorrent client maintains connections to a variable list of peers (a.k.a. nodes), and will upload to $k+1$ of them. The set of k is recalculated every M seconds, while every Q seconds an additional peer is unchoked in an attempt to find a better trading partner.

4.3 Incentives in BitTorrent

Is the strategy space provided by the BitTorrent client sufficient to be robust to rational manipulation? Will a rational user perform the default strategy by executing the BitTorrent client?

Before setting out to answer these questions, it is important to acknowledge the fact that BitTorrent is a popular system that leaves many users satisfied. The reality is that many BitTorrent users are unaware of the workings of the client software, and either view the cost of analyzing and modifying BitTorrent as too high, or more likely are simply unaware of the opportunity to express a different strategy. These users are *obedient* users, or perhaps could be modeled as *bounded rational* nodes.

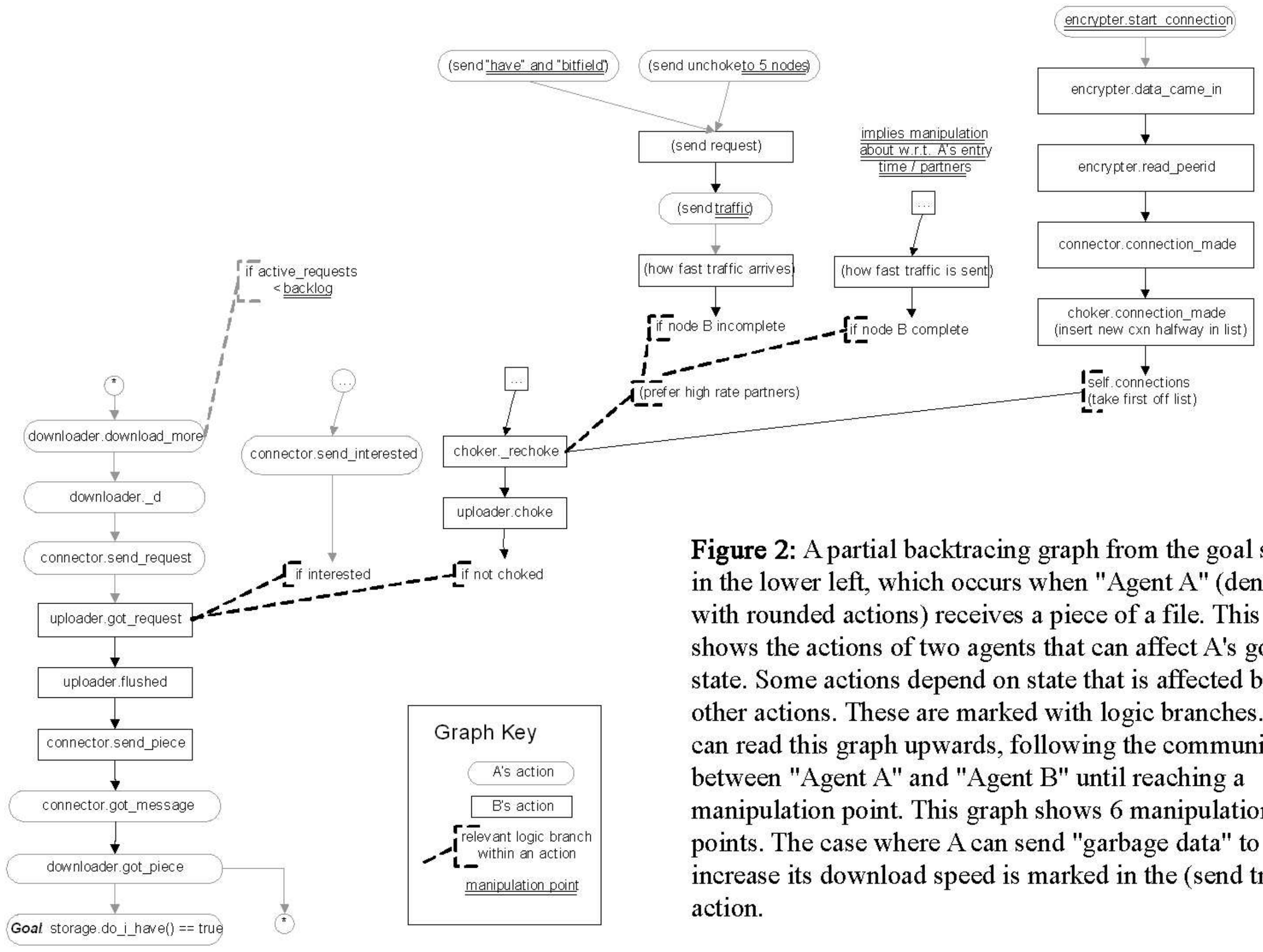
But, consider BitTorrent as run by a *rational* user who plays a strategy to maximize their own utility. Even understanding how a rational node would react to the BitTorrent client is a bit tricky, because there are different types of rational behavior. This highlights a problem in reasoning about faithfulness; one must state the assumptions made about the node's local utility models. For the purposes of this analysis, we will consider two simple types of rational nodes that can exist in the world:

- **Speed Critical:** This node places a high utility on receiving a file as fast as possible.
- **Free Rider:** This node places a high utility on receiving the file using as little upload bandwidth as possible. This models the case when people pay for their Internet connection by the amount of bytes transfered.

Although these seem like reasonable classes of peers, there are many other types of rational nodes. For instance, some people tend to start downloads before going to bed. These nodes probably have the same utility for receiving the file at all time steps until "tomorrow morning". Other nodes may be charged more money by their ISP for continued use of bandwidth above a given threshold for some amount of time. Though considering multiple node valuation functions is useful, we can demonstrate faithfulness problems in BitTorrent focusing on these two utility models.

A general assumption made in economics and multi-agent systems is that a node aims to maximize the benefit that it receives from the system. In Section 6.1 we consider extensions where peers also have anti-social [3] and malicious components of their utility function.

Our interest is in determining the faithfulness of the BitTorrent client specification for Speed Critical and Free Rider type nodes. As there is no formal specification for the Bit-



encrypter.start_connection
encrypter.data_came_in
encrypter.read_peerid
connector.connection_made
choker.connection_made (insert new cxn halfway in list)
self.connections (take first off list)
(send "have" and "bitfield")
(send unchoke to 5 nodes)
(send request)
(send traffic)
(how fast traffic arrives)
implies manipulation about w.r.t. A's entry time / partners
...
(how fast traffic is sent)
if node B incomplete
if node B complete
(prefer high rate partners)
if active_requests < backlog
*
downloader.download_more
downloader._d
connector.send_request
uploader.got_request
uploader.flushed
connector.send_piece
connector.got_message
downloader.got_piece
Goal storage.do_i_have() == true
connector.send_interested
if interested
choker._rechoke
uploader.choke
if not choked
Graph Key
A's action
B's action
relevant logic branch within an action
manipulation point

Figure 2: A partial backtracing graph from the goal state in the lower left, which occurs when "Agent A" (denoted with rounded actions) receives a piece of a file. This graph shows the actions of two agents that can affect A's goal state. Some actions depend on state that is affected by other actions. These are marked with logic branches. One can read this graph upwards, following the communication between "Agent A" and "Agent B" until reaching a manipulation point. This graph shows 6 manipulation points. The case where A can send "garbage data" to increase its download speed is marked in the (send traffic) action.

Torrent system, we analyzed the source code to understand the working of the system and look for strategy issues.

5. FINDING FLAWS IN FAITHFULNESS

Proving faithfulness for general algorithms is hard. Concurrent work explores one technique that can be useful in proving faithfulness [11]. That approach splits a distributed algorithm into disjoint phases, each of which can be proven faithful by showing that a node cannot benefit from any combination of deviations from the suggested strategy relevant to that phase. Phases are then certified and locked, so that phases cannot affect each other.

Demonstrating a system's faithfulness flaws is easier, but is also non-trivial. Faced with a specification (or source code), how can one find possible *manipulation points*?

5.1 Backtracing

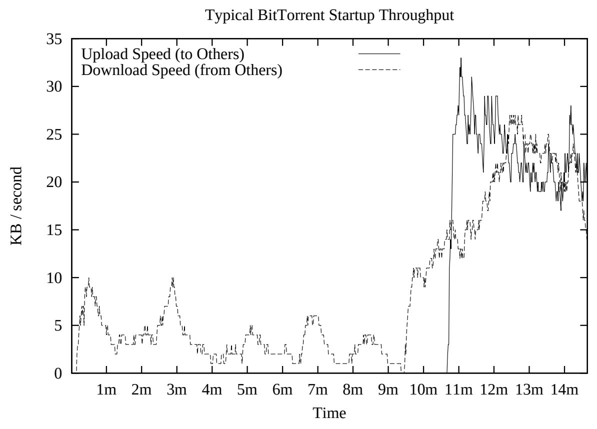
One idea that we found useful in studying BitTorrent is *backtracing*. The idea of backtracing is as follows:

```
For each type of rational node in the system:
  For each communication configuration:
    For each goal state:
    Trace backwards through logical steps (code)
     in interaction with others. Mark branches
     and neighbor interactions as candidate
     manipulation points.
    Examine and classify point.
```

As one works backwards through the code paths to find points in the code that affect this goal state, these candidate manipulation points are examined manually to explore the effects of a selfish manipulation. Each manipulation point can be be classified as one of the following:

- **False alarm:** A point that does not actually affect the path to the goal state.
- **Beneficial manipulation:** A point where a manipulation from the suggested strategy can occur, but is beneficial to the system goal. These points represent optimizations that should have been provided by the system designer.
- **Harmful manipulation** A point where a manipulation from the suggested strategy can occur that increases this node's utility, but hurts either the distributed mechanism execution or other nodes in the system.

Backtracing is similar to program slicing [22], where a slice consists of the parts of a program that potentially af-



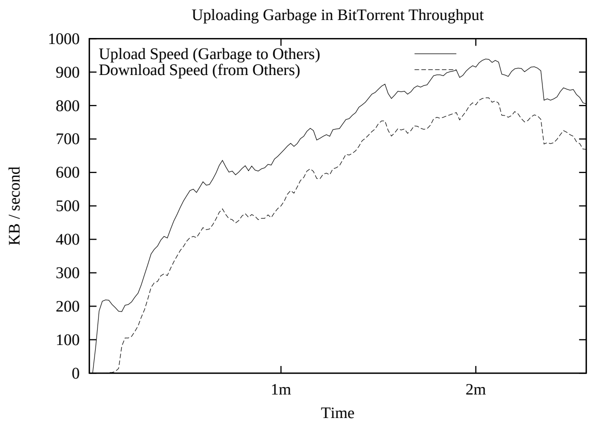


Figure 3: An illustration of how uploading garbage can help a Speed Critical rational node. On the left is throughput of a new peer entering a BitTorrent exchange. This peer benefits from optimistic unchoking from other nodes for about ten minutes. (Spikes occur because different nodes unchoke this node for some time, and then re-choke this node again. Just before 11 minutes, this node has built up a piece wanted by other nodes, and has begun uploading this data to other nodes. As a result, these nodes reciprocate, and the download speed climbs. Contrast this to the right-hand graph. First, notice the time and throughput scales are radically different; the maximum download speed of 30K/s in the left graph would show up as a small blip in the right graph. In the right-hand graph, the same node has falsely advertised having all pieces and is uploading garbage to other peers as fast as requested. Shortly after the uploading begins, the download speed (of good data) begins to increase as well.

fect the values computed at some point of interest [21]. One difference is that while program slicing considers one user in a single program, backtracing focuses on at least two users interfacing with the same client software in a game-like setting.

A sample backtracing graph between two peers in the BitTorrent system, generated by hand, is given in Figure 2. Each node in this graph represents a particular function or criteria that has the potential to affect the goal state. One can follow the function calls backwards in an attempt to find manipulation points. For each manipulation point, one must categorize the chance for manipulation as a false alarm, a beneficial manipulation, or a harmful manipulation. Manipulation points must be considered as set; a joint manipulation possibility exists if a node can deviate from the suggested strategy in multiple places to achieve a better outcome. This graph was generated with the Speed Critical rational user in mind. The goal of this user is to cause the goal state in the lower left (indicating that a piece has been downloaded successfully) to become true for all pieces as quickly as possible.

Nested for-loops, let alone the tracing step, suggest that this procedure can be onerous for large systems. It is labor-intensive to classify a candidate manipulation point and to verify that additional points (in the form of newly inserted branch statements) aren't valid manipulations. (Analyzing the BitTorrent system from scratch with two types of rational nodes, one goal state, and four possible communication configurations took us approximately eight hours.) Backtracing is imperfect, since it requires a human to analyze each candidate. The technique can only disprove system faithfulness and identify the design flaw. If no flaws are found, the system may or may not be faithful.

Despite all of these drawbacks, we have found the backtracing procedure to be useful. What is notable about the backtracing procedure is that *none* of the manipulations described in the next section were expected when we began this project, and one that we did suspect (declaring held pieces to manipulate connections) turned out not to be a false alarm, not affecting the BitTorrent protocol.

Most current incentives and mechanism design work is contained in relatively small systems like BitTorrent, and so the manual intensity of backtracing may not be too horrific. While it's by no means a perfect tool, backtracing can be a useful first-line faithfulness debugger.

5.2 Backtracing BitTorrent

In this section, we report the results of our backtracing exercise. The first set of results are information revelation issues, and do not change the underlying BitTorrent algorithm.

5.2.1 Identity

If a client announces itself multiple times to the tracker (perhaps under different port numbers), it appears to the system as a new client. The effect of increased visibility is that a node will be optimistic unchoked proportional to its declared identities. This is related to the well-known Sybil attack [5].

5.2.2 Upload Bandwidth

Because of the trickle-down nature of BitTorrent, a node that physically has higher bandwidth tries to ensure that it stays in the unchoked list of a partner, but it may be able to do this while appearing to be a slower than it is in reality. If a partner unchokes the fastest k nodes out of a list of $k+1$ potential nodes (see Figure 1), this peer must upload at a rate just above the kth fastest competitor.

Thus, the reported upload rate (really, the upload rate from this peer as measured by the partner) can be less than this peer's true speed.[1]

The next set of issues are algorithm manipulation issues.

5.2.3 unchoke_next

The BitTorrent client algorithm adds a new connection halfway into the unchoke_next list (See Figure 1) of the connection recipient. This means that a node that is newly choked may be better off closing and re-opening a connection to achieve a better place in the optimistic unchoking line.

5.2.4 Garbage Upload

A client can claim to have all pieces of a file, sending garbage to a peer that requests any piece. Although the garbage is detected by the peer through a data integrity hash check before writing to disk, the transited garbage counts as valid incoming traffic. A node with a high bandwidth connection can bootstrap very quickly with this attack.

An example of the benefits of such a manipulation was confirmed experimentally and shown in Figure 3.

5.3 Harmful Manipulation Summary

In summary, here are the *harmful manipulations* that the backtracing graph helped reveal:

- A Free Rider can run a Sybil attack on the tracker, claiming to be many parties (running on different ports on the same IP address). This increases the distribution of that peer's contact information to other peers, which in turn increases the likelihood of receiving pieces through optimistic unchoking.
- Once a Free Rider has been optimistically unchoked by a peer, sent a piece, and and then re-choked, it is moved to the end of that peer's unchoke_next list. The Free Rider can break and re-create the connection to jump in line to a more beneficial spot.
- The Speed Critical node can additionally play the Garbage Upload strategy.

The cost of each of these manipulations is less than 100 lines of Python code, and several hours of programmer time.

5.4 TCP Manipulations

BitTorrent, like many other distributed systems, is built on top of standard networking building blocks. BitTorrent runs on top of TCP. Are there strategies available to the BitTorrent user that are enabled as a result of changing the behavior of their TCP implementation?

While the cost of this modification is relatively high compared to changing a few lines of Python application code, the answer is yes. Without casting it as *rational* behavior, previous work in networking has established manipulation points in the TCP protocol [19]. The authors of that work did their analysis by instinct [20], focusing on message sequences that seemed problematic. That analysis could be further enhanced by a backtracing exercise on the TCP protocol. In a separate work, TCP issues are considered with other protocols, and design guidelines are given that might prevent the accidental creation of manipulation points [1]. That work is applicable to designers seeking to create faithful systems.

6. OPEN QUESTIONS

6.1 Beyond Rationality

In studying BitTorrent, we have limited our analysis to a network consisting of rational and obedient nodes. But there are other node types as well. Even though we have shown BitTorrent not to be faithful, we can speculate about how the system would work with nodes whose utility functions have not been addressed.

There are disadvantages to using a system like BitTorrent in the presence of peers with unanticipated utility functions. Antisocial [3] and malicious peers are a particular challenge. One issue is in how a peer is given the IP addresses of other downloaders. A malicious user could publish a file with a Trojan horse, and then generate a bootstrap list of probable affected targets from the list of IP addresses passed back from the tracker. Alternatively, if someone wishes to disrupt a file distribution, they can perform a Sybil attack [5], filling the tracker with clients who slow down distribution by providing garbage or no service.

More generally, what happens to mechanism design as we allow obedient, rational, and faulty nodes in the system? Computer science tends to treat entities in a network as obedient, or as a set of nodes that could be a obedient or some degree of faulty. Economics and mechanism design tends to treat entities as rational, or as a set of nodes consisting of rational and obedient (such as an obedient center.) What happens when these world-views collide? We posit two questions in this vein:

Open Question #1: There are well-known impossibility results in distributed systems. For instance, for cases of agreement in the presence of f faults, $3f + 1$ machines are requirement to correctly reach consensus [13]. What if a subset of the faulty nodes are actually rational nodes that are unable to express their preferred strategy with the default software? Are there cases where a rational node can be incented to perform correctly, moving the rational node into the "correct" category from the "faulty" category, thereby side-stepping the initial $3f + 1$ assumption?

Open Question #2: Start with a network consisting purely of rational nodes running a faithful mechanism. Add one Byzantine faulty [13] node that can act arbitrarily. How does this change the faithfulness properties? Can one design mechanisms that are faithful even in the presence of faulty nodes? Does the answer to this question change if you started with a network consisting of $n - 1$ obedient nodes, and 1 rational node?

[1]One could imagine a Vickrey-inspired scheme replacing this system, where a node reports its true upload bandwidth but is made to send at only the kth speed. Overstatements of bandwidth are prevented by throwing a node out of the system when it can't meet a bandwidth. Pragmatic issues make this complicated, since hidden bottlenecks between a source and sink may reduce the actual throughput, and an excluded node may be able to re-enter the system with no cost [5].

To our knowledge, nobody has built an analysis of any mechanism to demonstrate that it can operate successfully in the presence of rational, obedient, and faulty nodes.

6.2 Automated Tracing Tools

We wonder if the same sorts of systems tools that have been developed to aid people in program slicing can be useful in faithfulness analysis.

Open Question #3: Can program slicing ideas/tools be applied in finding manipulation points in real systems?

Backtracing is less related to the ideas of model checking, where an implementation is checked for correctness against a specification. While this is an active area of systems research [6], it seems more likely that model checking tools would reveal general specification clarity issues. The resulting specification or implementation would need some form of manipulation analysis.

7. CONCLUSION

In this paper, we explored how software can be an expression of node strategy, and how faithfulness relates to rational manipulation of this strategy. We examined a particular file download system called BitTorrent and used backtracing, a technique similar to program slicing, to find manipulation points. After categorizing the manipulation points in BitTorrent, we found several harmful manipulations that negatively affect the welfare of other nodes in the system. Some nodes may view the costs for manipulating BitTorrent in this way as relatively small compared to the potential gain. Finally, we began to explore some open problems relating to showing faithfulness in algorithms to be run on the Internet.

8. ACKNOWLEDGMENTS

Thanks to Peter Key, Margo Seltzer, Danni Tang, and anonymous reviewers for their useful comments. This work is supported in part by NSF grants IIS-0238147 and ACI-0330244. This research was performed while the first author was an intern at Microsoft Research Ltd.

9. REFERENCES

[1] T. Anderson, S. Shenker, I. Stoica, and D. Wetherall. Design Guidelines for Robust Internet Protocols. *HotNets-I*, October 2002.

[2] Blizzard Game Web site, World of Warcraft BitTorrent FAQ. http://www.blizzard.com/wow/faq/bittorrent.shtml.

[3] F. Brandt and G. Weiß. Antisocial agents and Vickrey auctions. In J.-J. Ch. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, volume 2333, pages 335–347. Springer, 2001. Revised papers from the 8th Workshop on Agent Theories, Architectures and Languages.

[4] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer to Peer Systems*, June 2003.

[5] J. Douceur. The Sybil Attack. In *"1st Int. Workshop on Peer-to-Peer Systems (IPTPS'02)"*, 2002.

[6] Dawson Engler and Madanlal Musuvathi. Model-checking large network protocol implementations. In *To appear in NSDI 2004*, 2004.

[7] Joan Feigenbaum, Christos Papadimitriou, Rahul Sami, and Scott Shenker. A BGP-based mechanism for lowest-cost routing. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, pages 173–182, 2002.

[8] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, 2002.

[9] Info-Anarchy Web site, definition for Parallel Downloading Systems: Hive. http://www.infoanarchy.org/wiki/wiki.pl?hive.

[10] Jeffrey Shneidman and David C. Parkes. Rationality and Self-Interest in Peer to Peer Networks. In *2nd Int. Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[11] Jeffrey Shneidman and David C. Parkes. Specification Faithfulness in Networks with Rational Nodes. In *Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004)*, July 2004.

[12] Lindows Web site, Press Release about BitTorrent. http://www.linspire.com/lindows_news_pressreleases_archives.php?id=111.

[13] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[14] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 129–140, 1999.

[15] Noam Nisan and Amir Ronen. Computationally feasible VCG mechanisms. In *Proc. 2nd ACM Conf. on Electronic Commerce (EC-00)*, pages 242–252, 2000.

[16] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 35:166–196, 2001.

[17] David C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency (Chapter 2)*. PhD thesis, Univesity of Pennsylvania, May 2001.

[18] Pablo Rodriguez. *Scalable Content Distribution in the Internet*. PhD thesis, Federal Institut of Technology, Lausanne (EPFL), Sept 2000.

[19] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[20] Stefan Savage. Private Communication, 4/2004.

[21] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[22] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.