CSE461 Project 3: Tor61

Out: Wednesday February 12, 2014
Due: Monday March 3, 2014 by **11:59pm**.
Teams Allowed: Yes
Teams Encouraged: Yes
Ideal Team Size: 2

Summary

Tor is an overlay network whose goal is to provide anonymity. When used to carry browser traffic, Tor makes it very difficult to determine which sites the user is visiting by looking at packets carried by the Tor network, or to determine who is visiting a site by looking at the traffic between the Tor network and the site. Tor works by routing traffic from the browser through a number of Tor router nodes before sending it to the web server. Pairs of routers communicate over TCP connections, so at each hop of the route the source IP addresses in packets name the source Tor router of that hop. Encryption is used to hide both the browser's and the web server's IP addresses, so that not even the Tor routers can make an association between the browser and the site it is visiting. (Additional Tor references are on this course page.)

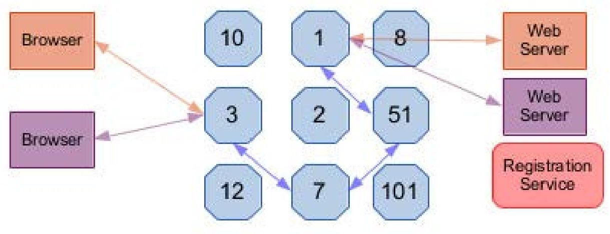
Tor61 is an overlay network whose architecture and protocol is based on Tor. The main simplification in Tor61 is that no encryption is used. That substantially reduces the kinds of anonymity attacks that it can defend against, compared with Tor. But, it's still the case that most Tor61 routers carrying a browser's traffic, as well as the web site, cannot determine the IP address of the browser. Additionally, while Tor61 carries general TCP traffic, we provide a protocol converter only for HTTP, and our only intended client application is a browser.

This project will provide a number of challenges:

- The Tor61 architecture may take a bit of effort to understand. Translating that architecture into code can be even more challenging. Careful design and planning will probably be more important than you're accustomed to.
- Tor61 is inherently distributed, in a manner more complicated than client-server. Debugging can be frustrating. Writing debugging/monitoring/logging code is probably essential, even though that code is not part of the assignment requirements.
- Ultimately all of our implementations need to inter-operate. Distinctions among the implementations (e.g., different timeout values) can cause two implementations that work fine on their own to fail when they try to inter-operate.
- Our ideal goal is to bring up an instance of each team's Tor61 router and leave it up, for weeks. Writing code that stays up for weeks is harder than code that can handle a few requests.

Tor61 Architecture Overview

A Tor61 network consists of a set of Tor61 router nodes, shown in the figure below in blue. When a router comes up, it contacts the registration service to register itself and to fetch the IP:port addresses of other (hopefully) online routers. Each router creates a Tor61 *circuit*, a random path through the Tor61 routers. Together, the three blue lines in the figure below represent the circuit created by router 3. Each individual link is a TCP connection between a pair of routers. The routers speak to each other using the Tor61 protocol (described in a bit), which is carried over the TCP links.



Tor61 routers also act as HTTP proxies. A browser's HTTP request is routed over the Tor61 circuit created by the router acting as the browser's proxy. That router (3 in the figure above) issues the Tor61 protocol packets that carry the HTTP request to the router at the other end of the circuit (1, in the figure), which reconstructs it as an HTTP stream and passes it to a web server. The response from the web server to the browser is carried back using the same process.

Each HTTP request creates a Tor61 *stream*. Logically, a stream is a TCP connection between a browser and a web server that has been cut in the middle and had a Tor61 circuit spliced in. In the figure we show two streams, representing two requests issued by two distinct browsers and going to two distinct web servers, but concurrent HTTP requests by a single browser would also create multiple streams, whether or not they go to distinct web servers.

Note that both streams in our figure are carried over the same circuit. This means that a single TCP connection is carrying Tor61 protocol cells for possibly many different streams at once. When those cells reach their destination (say, router 1), they must be *demultiplexed* - unentangled and delivered to their appropriate end point, one of the two web servers. Similarly, the two streams of web server responses are carried over the same circuit, and so the same TCP connections, and their originally distinct traffic, must be *multiplexed* onto the circuit and then demultiplexed for delivery to the correct browser.

As well as multiplexing streams on circuits, Tor61 multiplexes circuits on TCP connections. Each pair of routers creates at most one TCP connection connecting them. If two or more circuits include that pair of routers as a hop, those circuits share that TCP connection. For example, it's not shown in our figure, but suppose that router 10 has created the circuit whose path is routers 10, 2, 3, and 7. In that case, the circuits for routers 3 and 10 share the TCP connection between routers 3 and 7; i.e., that one TCP connection multiplexes the traffic of both circuits.

Tor61 Cells

Tor61 routers communicate using *cells*, which are fixed-length packets. Tor61 cells are always 512 bytes long. If some cell doesn't require 512 bytes, it is padded (with zeroes, say) to be exactly 512 bytes. If some data needs to be moved and is longer than 512 bytes, it is sent in a sequence of cells. Fixed-length cells are important to Tor for anonymity preservation. We maintain them in Tor61 because they simplify demultiplexing of incoming data: we know that each cell is a consecutive chunk of 512 bytes, so we don't have to parse the incoming TCP stream to break it up into cells.

There are basically two types of Tor61 cells, distinguished by their destination on the circuit. *Control cells* have the next hop router as the destination. *Relay cells* have the router at the end of the circuit as their destination. Both control and relay cells are used to create the circuit. Relay cells are used to carry data between the browser and web server. Control cells contain (minimally) a circuit number and a cell type field. Relay cells contain additionally a stream number and a relay cell type field.

Cell Formats

All cells are 512 bytes long. Sufficient padding is added to the end, if necessary. The contents of the padding are irrelevant to the encoding.

All cells begin with a common header consisting of a two-byte circuit id value and a one-byte unsigned integer cell type number. All multi-byte numeric values are in network order. We do not show the padding required to bring the cells up to 512 bytes.

Cell formats and use are, for the most part, identical to those in Tor, so Tor documentation may provide additional useful information. The primary changes are the addition of some cell types (Open, Opened, Open Failed, Create Failed, Relay Begin Failed, and Relay Extend Failed), which helps simplify cell processing. Additionally, we include but do not use the digest field of the Relay cell header, which in Tor is a cryptographic message digest used to ensure end-to-end integrity of the data stream .

Open

0x0000	0x05	agent id of opener end	agent id of opened end
2	*1*	*4*	*4*

When one agent (the opener) opens a TCP connection to another (the opened), it must immediately send an `Open` cell to identify the participants on that connection. Successful response is an `Opened` cell; failure response is a `Open Failed` cell.

Opened

0x0000	0x06	agent id of opener end	agent id of opened end
2	*1*	*4*	*4*

This response indicates that the opened agent accepts the newly opened connection. Note that the cell contents are identical to the `Open` to which it responds, except for the cell type header field.

Open Failed

0x0000	0x07	agent id of opener end	agent id of opened end
2	*1*	*4*	*4*

Failure response to an Open cell. Note that the cell contents are identical to the `Open` to which it responds, except for the cell type header field.

Create

circ id	0x01
2	*1*

Requests establishment of a new circuit, with id given by the header field value. Success response is a `Created` cell; failure is a

Create Failed.

Created

circ id	0x02
2	*1*

Success response to a `Create` cell.

Create Failed

circ id	0x08
2	*1*

Failure response to a `Create` cell.

Destroy

circ id	0x04
2	*1*

Requests teardown of the circuit whose id is given by the header field value. There is no response.

Relay

circ id	0x03	stream id	0x0000	digest	body length	relay cmd	body
2	*1*	*2*	*2*	*4*	*2*	*1*	*<body length>*

A `Relay` cell encapsulates a number of distinct types of subcells. We show in the diagram the header for the subcells, shared by all the `Relay` subcell types. Some of those types carry additional data in the `body` field. The `relay cmd` field identifies the type of the subcell.

The subcell types are:

Name	Relay cmd	Body	Use
Begin	0x01	ip:port\0 (A null terminated string)	Create a stream (on the circuit named in the header). A TCP connection to the ip:port given by the body data should be opened and associated with the stream.
Data	0x02	stream data	The cells are used to move data between the two TCP connections at the ends of the stream.
End	0x03	None	A request to close a stream.
Connected	0x04	None	A success response to a Begin.
Extend	0x06	ip:port\0<agent id> (The address is a null-terminated string. The agent id is a 4-byte unsigned int.)	Used to extend a circuit. The current end of the circuit should send a Create cell to the agent named by the body data. Note that, unusually, this is an operation on the circuit.
Extended	0x07	None	Success response to an Extend cell.
Begin Failed	0x0b	None	Failure response to a Begin cell.
Extend Failed	0x0c	None	Failure response to an Extend cell.

Circuits & Routing

Overview

At the highest level, Tor61 routing works like this:

- First, establish a circuit - a path through the Tor61 routers. As part of this, each router on the path maintains a routing table that tells

it what to do with cells that arrive on particular circuits. For instance, router 7 in the figure above would have a routing table entry indicating that cells on the circuit sourced by router 3 should be sent to router 51.
- Now create a stream on that circuit. A stream connects a browser with a web server, and is associated with a single circuit.
- Now carry traffic from between the browser and the server by encapsulating the data stream in Tor61 Relay Data cells. Each cell carrying data contains a circuit number and a stream number. The routers use the circuit number to look up what to do with the cell in the routing table. The endpoint routers use the stream number to determine which browser or web server TCP connection to send the cells data on.

That's the general idea: once the circuit is established, cells carry the circuit ID, rather than an explicit destination address, and routing uses the circuit ID to look up an entry in a routing table.

Globally Unique Circuit IDs?

For this to work exactly as described, all circuits, everywhere in the Tor61 network, must have unique identifiers. Arranging for that is very hard. It would be very expensive (in terms of performance) to have all nodes agree on who gets to use circuit number 7, say. It's much easier if we can allow each router to pick its own circuit numbers, and it turns out we can. (One way would be to statically partition the circuit number space and give each router a unique subrange of circuit numbers. That would work, but isn't very flexible, so we use another approach, the one used by Tor.)

The key is to realize that circuit numbers don't have to be globally unique; we require just uniqueness on each hop. In the figure above, the blue circuit could have number 10 on the 3-7 hop, and number 21 on the 7-51 hop, and number 12 on the 51-1 hop, so long as no other circuit uses those numbers on those hops. Router 7's routing table would now say "If I get a cell from router 3 on circuit 10, I should forward it to router 51 and change its circuit number to 21."

Circuit Creation

A circuit is created like this:

- The "source router" picks a next router at random. If it has an existing Tor61 connection with that router, it uses it. Otherwise, it opens a TCP connection and sends an Open cell. If successful, an Opened cell should be received within a timeout interval. It has now established a Tor61 connection with the other router (and vice versa).
- The source router picks a circuit number, `C`, for the new circuit that is unique between the source router and the next hop router. It sends a Create cell with `C` as the circuit number in the header. If successful, a Created cell is returned.
- To extend the route by one cell, the source router sends a Relay Extend cell on circuit `C`. That cell contains as data the ip address and port of the desired next hop router to which the circuit should be extended. The last router on the current circuit processes the Relay Extend:
    - If it has a Tor61 connection to the target next router, it uses it. Otherwise, it creates one (opens a TCP connection and goes through the Open/Opened protocol).
    - It now chooses a circuit number, `C'`, that has not been used on the connection between it and the new target router, and sends a Create cell with circuit `C'` on it. If it gets back a Created cell, it enters in its routing table an entry that says "Cells received on circuit `C` from the source router should be forwarded to the next hop router on circuit `C'`" and sends a Relay Extended cell back to the source router.
- The source router repeats the Relay Extend protocol for each additional hop it wishes to add to the circuit. If an extend fails for some target, it eliminates it from the set of potentially useful routers and continues trying to find a router to which the circuit can be extended.

Stream Creation

Once the circuit is established, it can be used to create a stream connecting the browser and web server:

- When a browser connects, the header of the HTTP request is processed to identify the ip:port of the web server.
- A stream number, `S`, that is not in use on the source router's circuit is chosen.
- A Relay Begin cell is sent with circuit number `C` and stream number `S`. It also contains the ip:port of the web server.
- The last router on the circuit receives that cell and attempts to establish a TCP connection with the web server. If successful, it sends a Relay Connected cell back to the source router.

Routing Data

Finally, the HTTP data can be carried between the two:

- The source router packages the stream of data received from the browser into Relay Data cells and sends them on circuit `C` and stream `S`.
- The router at the other end of the circuit receives those cells, extracts the data from them and sends that data on the TCP connection it has for stream `S`, which delivers the data to the web server.
- The same process, with roles reversed, is used to convey the web server's response to the browser.

Architecture Topics

Registration Service Use

The registration service associates a string name and an integer data item with a registered IP:port. We use the name field to allow us to select either all routers, no matter who wrote them, or just those written by some specific group (for example, your own!). The registration name should have format `Tor61Router-xxxx-yyyy`, where `xxxx` is a group number, unique to your group, and `yyyy` is an instance number that allows your group to run more than one router copy at a time. As a string, your group number should be four decimal digits, including leading zeroes. To interact with only your own routers, issue a fetch requesting name prefix `Tor61Router-xxxx`. To operate with all extant routers, use prefix `Tor61Router`.

The registration service data field is used to hold a Tor61 router number, which is the name used for the router in some Tor61 cell formats. The router number is the 32-bit value `(xxxx << 16) | yyyy`, i.e., the binary concatenation of the group and instance numbers.

For this to work, we need to assign unique group numbers to all groups. That's a distributed coordination problem. We'll use a classic, distributed approach to solving it: everyone picks a group number at random and hopes for the best. (If you notice you've collided with another group(s), use a sensible collision resolution protocol to resolve it!) There is one well-known group: 0001 is reserved for routers run by the course staff.

Circuit Establishment

In Tor61, each router creates a circuit when it comes up. This differs from Tor, which (basically) allows each client to select a circuit. Tor provides more reason to care what your circuit is than Tor61 does, though.

Tor61 circuits have a fixed length: they include three router-to-router TCP connections. The routers used in a Tor61 circuit are chosen at random. The Tor61 specification is silent about whether or not a single router can appear in a circuit's path more than once, meaning that it's allowed but not required. Tor highly discourages doing that, although it's not clear that the specification actually forbids it. Allowing a router to appear more than once simplifies the Tor61 code and allows you to debug by bringing up a single Tor61 router instance that creates a circuit through itself (three times). Note that because other router implementations may place your router on a single circuit multiple times, your router needs to work in that situation, even if you never create a circuit with repeated router entries yourself.

Tor uses a circuit only for about a minute, then creates a new circuit on which all new streams are routed. In Tor61 we use a circuit until it appears to no longer function. When we think the circuit has become useless, we create a new one.

Circuit Number Assignment

There is a potential race condition in assigning circuit numbers, even on a single Tor61 connection: the two routers on the connection may nearly simultaneously decide to use the same circuit number `C` to create a new circuit. To avoid this, we adopt this version of the rule used in Tor:

> The router that initiated the TCP connection assigns only odd circuit numbers on that connection. The router whose TCP server socket was contacted assigns only even circuit numbers on that connection.

Stream Number 0

Stream number 0 is reserved for, and should be used for, Relay Extend cells and their responses, Relay Extended and Relay Extend Failed. These cells are used for circuit management, and are not part of any stream.

HTTP Proxy / Tor61 Interface

We include HTTP proxy functionality as part of the implementation of a Tor61 router. That allows the proxy to make direct calls to router code. (Tor separates these components, and communicates between them using a network connection.)

This means that each Tor61 router has a number of different kinds of sockets:

- A single TCP server socket waiting for connections from browsers.
- Possibly many TCP sockets returned from `accept` on the server socket. HTTP passes between the proxy functionality portion of the router and the browser over these connections.
- A single TCP server socket waiting for connections initiated by another Tor61 router.
- Possibly many TCP sockets connected to other Tor61 routers. These may have been returned from `accept`, if the other router initiated the connection, or may have been created by this router and connected to the other router as a client. The routers speak the Tor61 protocol over these sockets.
- Possibly many TCP client sockets connected to web servers. A new TCP connection is established each time the router processes a Relay Begin cell.

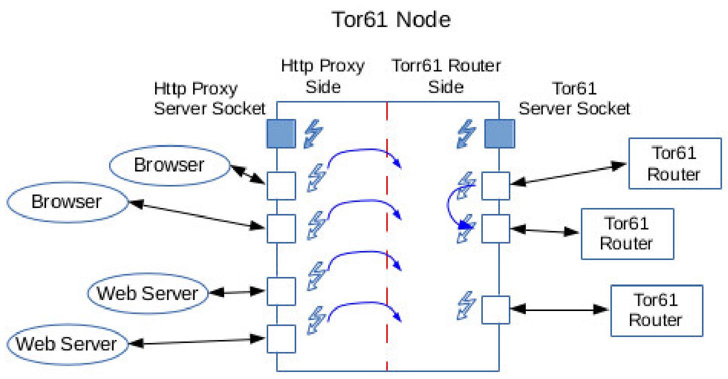
Threading & Distributed Deadlock

Note: This section suggests an implementation approach. You are not required to use this implementation. The implementation approach is not part of the protocol specification.

The potential for distributed deadlock is one of the two most complicated issues in the implementation. It's not so bad if you have a well-defined approach when you start coding, but can be very frustrating if you try to design as you code.

Basic Threading Architecture: Moving Data

Let's start by looking at "the natural" use of threads in the router, and restrict our attention to what it required to move data over the Tor61 network between browsers and clients. This picture is incomplete, but we'll refine it in a minute.

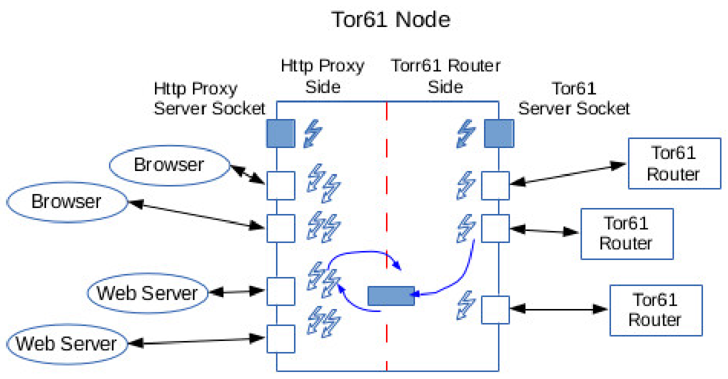


The large rectangle is a node, implemented by your code. It contains a part that acts as an HTTP proxy (actually, an HTTP-Tor61 protocol converter) and a part that performs Tor61 router functions. The small rectangles are sockets. The two solid rectangles are server sockets, waiting for incoming TCP connections. The other sockets are established TCP connections.

Associated with each TCP connection is a thread, shown as a lightning bolt, used for reading from the connection. Proxy-side threads take what they've read, package it up in Tor61 cells, and then invoke a Tor61 send function to send those cells. Router-side threads read incoming cells and act on them. An important activity is routing the cells to the next hop of their circuit. In that case, the reader thread invokes a send operation on the next hop's TCP socket to move the cell on its way, then goes back to reading.

One thing missing from our first-cut diagram is the flow of data from the Tor61 router side back to the proxy side. For instance, how does data that comes out of the Tor61 network make its way onto the TCP connection with the web server? We require an additional thread per proxy side TCP connection for this, one that sits in a loop reading from Tor61 network and writing to the TCP socket. But, exactly how can that work? This new thread can't actually read from a Tor61 socket, because it might end up reading some cells not intended for it.

The solution is to provide some kind of buffer between the router and proxy sides. When the socket-reading thread on the router side reads a cell on a circuit that ends with its node, it determines what stream it is on, and from that what buffer to place the cell into. It then goes back to reading from the Tor61 socket. The proxy side reader cell now takes the cell out of the buffer and writes its data to the TCP connection. This figure shows this additional complexity:



The thread that reads the proxy's TCP connection operates as before, reading from the socket and then invoking router-sider routines to send the data as Tor61 cells. The newly added proxy-side "writer thread" reads from a buffer shared between it and the Tor61 router side. The Tor61 socket-reader threads put cells into the buffer when they determine that the cells belong there - the circuit ends at this node, and the stream id of the cell indicates it belongs to the TCP connection whose buffer we'll put the cell into. Because many different threads may try to operate on the buffer at once, it has to be "thread safe." Java, C++, and Python (and perl) all provide threadsafe queue implementations in standard libraries.

Now we have data moving in both directions between the Tor61 stream endpoints (the proxy connections) and the Tor61 router connected to them, and between one Tor61 connection and another (for routing). We're still missing an important bit of functionality, though: how to block while waiting for a response.

Blocking for Sequenced Operations: Request-Response Exchanges

Some Tor61 cells are used to request control operations that require a response for successful completion. In particular, Open, Create, Relay Extend, and Relay Begin all require responses. This brings up the issue of how to connect the response cell with the original request cell, once the response arrives.

The approach we have been using is to use threads, whose control flow naturally encodes sequenced operations and current progress through them. For instance, Create expects a Created reply. We can link the two by having the thread that sends the Create block until a reply arrives, and then checking that it is a Created before finalizing the operation. Doing this requires that we solve three subproblems, though:

- What thread do we use to link the request and the response?
- How does that thread block? What does it block on?
- How do we avoid deadlock?

Which Thread?

A full discussion of this topic is an unfortunately large number of words, but the essential point is relatively simple:

> ***Never use a Tor61 "reading thread" (one whose main task is reading from a socket) to do anything that might cause it***

to block.

So, for instance, suppose a Tor61 reading thread reads in a Relay Extend cell and finds that its router is the circuit end point. That's a request for its router to send a Create cell to the router named in the Relay Extend, to wait for a Created response from that next router, and then to reply to the Create with a Created cell. The reader thread cannot take this on itself, because it would have to block while waiting for the Created reply. Blocking can lead to deadlock (described briefly in a bit). Instead, the reader thread should create a new thread and hand it the Relay Extend cell for processing.

On the other side of this coin, suppose the Tor61 reader thread reads in an Open cell. It can simply send the Opened reply, because writing isn't a blocking operation (in the sense we mean it here). Similarly, the reader thread can perform routing, and process any other kinds of cells when it's guaranteed no blocking will be needed.

So, the answer to "which thread?" is "some thread that isn't a Tor61 reader thread, if processing might block." In some cases you create a new thread just to handle the request. In others, you are executing with some thread when you realize you need to engage in a request-reply exchange. Depending on your implementation, you might just use that thread, so long as it's not a Tor61 reader thread. (A proxy reader thread is okay, though, for example.)

What to block on?

We use the same solution as we used earlier for the Proxy writer threads: a thread-safe buffer. In this case, the thread waiting for a Tor61 response cell tries to read from the buffer, and blocks until there is something to read. The Tor61 reader threads examine each incoming cell, and if they find one that is a response directed to this node, they insert it into the buffer where the calling thread is presumably waiting.

This simple scheme works so long as there can be no confusion over which responses correspond to which requests. Because neither requests nor responses carry sequence numbers (or any other kind of explicit, unique name), we have to rely on ordering. In particular, we use the fact that there is at most one outstanding operation being performed on any one circuit (i.e., at most one Open, Create, or Relay Extend), and at most one outstanding operation on any stream (i.e., at most one Relay Begin). That means that by having a buffer per (connection, circuit, stream) there can be no ordering issue. (Note that ordering is sufficient only because we're operating on top of TCP, which provides reliable streams, including duplicate detection.)

Deadlock

Deadlock is when a thread is waiting for something, but that something can't be provided until that thread does something, which it can't do because it's blocked waiting for that something. In this application, the main danger is that that router sends a request cell to another router, which simultaneously sends its own request cell back to the first router. In both routers the thread that sent the request is blocked waiting for a response. If the socket-reading threads on the two routers are the ones sending the requests, they're both blocked. Since neither can read the request the other has sent, neither can send a response, and so neither can ever receive a response.

Resource Leaks

It turns out to be a lot easier to bring up routers and circuits than to shut them down cleanly. Unless you're careful, it's easy to leak resources. Resources include not only memory but also threads and sockets. To avoid this, it's important to verify that your code meets at least a minimal standard, releasing all resources in simple cases, for example, both ends of a stream close their connections or a Destroy cell arrives (asking that a circuit be destroyed).

This problem is made even harder because it's difficult to know what you should do in unexpected cases. For instance, suppose you get an indication that no more input will be coming from the connection to a browser. Should you close that connection, or might there be additional data (perhaps in flight in the Tor61 network) to send to it? Can you even write that data to the browser if it has closed the output side of its connection? What if the browser or the server never close their connections? There are many behaviors that "should never happen" but likely will be observed, if you visit enough pages.

One possibly useful way to test that you're cleaning up is to verify that all resources have been released when you close the program gracefully. That means that all threads have terminated, all sockets have been closed, and all memory released. These things aren't really that important at shutdown, because at that point the resources will be released by the system if you don't do it, but a clean shutdown is an indication you've managed to keep track of all resources and have code that will deallocate them.

If you leak resources, eventually your process will become unresponsive. If you run out of sockets, for instance, no new connections can be made. Most CSE systems limit an individual process to 1024 open sockets, so if you load a few complicated pages while leaking sockets you can run out quite quickly.

Because of this difficulty, we establish three levels of functionality:

Tier III
Multiple copies of your program can run together and successfully fetch at least a few complicated web pages. You may hang after a bit, though.

Tier II
You can run even when interacting with correct implementations written by others. You continue to stay responsive for a medium amount of time. ("You have a small leak, not a gaping hole.")

Tier I
You can run in a heterogeneous environment and stay up for long durations -- days, say.

How We'll Grade & Grading

Basically, we'll try to determine which tier your implementation falls into. Tier 2.2 performance (Tier II with a few minor deficiencies) gets you full points. In addition to full points, Tier I performance gets your vast respect points from your peers and possibly even strangers.

We'll run our own registration server while grading. We'll first bring up multiple instances of your program in isolation, i..e, a Tier III environment. We'll test by connecting to the last node brought up (the one that sees the most other nodes already up when it starts). If that goes well, we'll try it when running with other instances of the sample solution (i.e., a controlled Tier II). Finally, if that goes well, we'll toss it into a Tier I sandbox with the other programs that have made it that far and check back on it in a couple days.

Testing

We strongly advise starting simple, not by trying to fetch cnn.com. For the simplest possible testing it's handy to use two instances of `nc`. Start one listening on a server socket; it will act as the web server. Connect the other to the HTTP proxy port of an instance of your router. It acts as the browser. Now type a valid HTTP Host line in the browser side, giving the IP:port address of the server `nc` instance. Type in the browser instance and see if it appears in the server instance. Type in the server instance and see if it appears in the browser instance.

Once that works, or if it gets too tedious, you might try using a real browser and web server. We've set up the simplest possible web page, one that just returns a tiny bit of HTML and doesn't require the browser to fetch any additional resources. That page is at http://courses.cs.washington.edu/courses/cse461/14wi/projects/proj3/simple.html.

`run` Script & Other Requirements

Your run script should implement the following invocation syntax:

```
./run <reg server host/ip> <reg server port> <group number> <instance number> <HTTP Proxy
 port>
```

An example invocation is:

```
./run 128.208.1.137 46101 3147 2 46110
```

This would mean that the registration server to use is at 128.208.1.137:46101, that the router's string name is Tor61Router-3147-0002, and that it should establish its HTTP Proxy on server port 46110.

Your program should print identifying information about the Tor61 routers used in the circuit it creates. It should be easy to spot that information; don't bury it among many many lines of debugging output. It should also be easy to interpret that information so that we know, for instance, what order the routers appear in on the circuit and which router instances are part of the path.

Turn in

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `proj3`. There should be no other files in that directory.
2. Create a README.TXT file that contains the names, student numbers, and UW email addresses of the member(s) of your team.
3. Put the README.TXT file, your project 3 solution source code, and your `run` script in the `proj3`directory.
4. While in the directory that is the parent of `proj3/`, issue the command `tar czf proj3.tar.gz proj3`.
5. Verify that the tar file contains the files you intend to submit: `tar tf proj3.tar.gz`.
6. Submit the `proj3.tar.gz` file to the course dropbox. (There's a link to the dropbox in the navigation section of all course pages.)

Appendix A: How to run for days

On at least most CSE systems (including the `attu`'s), you can create a shell session that will survive logout using the `screen` command. (User-typed input is shown in red.)

```
$ screen # starts a screen session
$ ./run .... # start up your program
Starting...
HTTP proxy is up on port 17344
Tor61 router is up on port 37288
...more program output...
ctrl-a d
[detached from 29680.pts-98.attu2]
$ # you're now back in your original shell, and your program is still running in the screen
session
$ # do anything, including logging out
$ # eventually come back
$ screen -r 29680.pts-98.attu2 # reconnect to the screen sesion
HTTP proxy is up on port 17344
Tor61 router is up on port 37288
...more program output...
 < you'll see whatever your program has output while you were away >
ctrl-a d
[detached from 29680.pts-98.attu2]
$
```

Your screen session will not survive a machine reboot, but those are rare. If you want to come back up automatically after a reboot, look into `cron` and `crontab`.

Appendix B: Cell exchange sequences

We show here typical request-response cell exchanges in the Tor61 protocol. This excludes cells that travel in one direction with no response, which includes the movement of Relay Data cells and the Destroy control cell.

Initialize a newly opened TCP connection

```
OPEN --------->
     <-------- OPENED
```

Establish first hop of a new circuit

```
OPEN   --------->
```

```
       <--------- OPENED
CREATE --------->
       <--------- CREATED
```

The Open/Opened is optional -- it's required, as a side-effect, if there is no existing Tor61 connection with the target Tor61 router. The Create/Created exchange is required to establish the new circuit, and is required even if a Tor61 connection already exists between the two routers.

Extend the circuit

```
RELAY EXTEND -----> [forward] -------->
                                         CREATE ------->
                                                <------- CREATED
            <----- [forward] <--------- RELAY EXTENDED
```

The Relay Extend cell is passed to the current end of the circuit. The router there engages in the Create exchange to extend the circuit by one hop, then sends a success or failure response back to the circuit's source node.

Create a stream

```
RELAY BEGIN -----> [forward] -------->  <open TCP connection to web server> ------->
            <----- [forward] <--------- RELAY CONNECTED
```

The Relay Begin is passed to the Tor61 router at the end of the circuit.