Dynamic Content on Information-Centric Networks and BitTorrent

Harry Yu

Abstract

The Internet has changed to become more content-focused. Many researchers have proposed Information Centric Networks (ICNs). These networks allow people to retrieve content by content names instead of host names and have the potential to replace Content-Delivery Networks (CDNs) and Peer-to-Peer (P2P) networks. This paper examines whether these ICNs can effectively serve dynamic web content. If ICNs could serve dynamic web content, that would unlock a huge number of applications, including MapReduce engines and video encoders with embedded data. ICNs bring quick scaling, intelligent routing, and automatic load-balancing to these clustered tasks.

I analyze several important areas needing improvement: routing of user provided input, system-resource-based routing, techniques for load balancing and rejecting requests, and support for verifying the output of untrusted peer execution systems. I also confirmed that the naming and routing schemes of NetInf and DONA are already compatible with self-verifying programs owned by specific users. I found that it’s easy to extend NetInf and DONA to accommodate dynamic content, but harder to extend PURSUIT/PSIRP and Named Data Networking architectures.

I conclude by making a case for BitTorrent as a base for building ICN technology onto. BitTorrent shares many elements of ICN proposals’ routing and naming systems, and has a large community of software to build onto.

I build and analyze a working prototype of a BitTorrent-based peer-to-peer execution system which executes a Hello World program in Python. From this prototype, I identify three main issues to tackle in a production client: security, resource management, and initial connection time.

Introduction

The Internet’s traffic and complexity have both grown immensely over the last ten years. Traffic growth on the Internet has been driven mainly by video-on-demand applications such as Youtube and Netflix, which together account for almost half of all Internet traffic.\(^1\) However, the Internet’s original host-based infrastructure often doesn’t lend itself to scalable, reliable, and cost-efficient solutions.

---

The new traffic demands have led to complex application-layer solutions to these problems. Youtube, Netflix, and other large online content providers use Content-Delivery Networks (CDNs) and Peer-to-Peer (P2P) networks to deliver their data. These technologies rely on caching and replication to provide quick, location-independent access to static data. Today, users don’t search for hosts. Rather, users search for data and services.

Researchers have proposed many Information-Centric Network (ICN) networks. These networks use Internet- and transport-layer architectures to solve the same problems. ICNs are networks where requests consist of information identifiers rather than host identifiers (IP addresses). They can be seen as a more sophisticated BitTorrent network.

ICN routers route requests to hosts with the requested content. Compared to application-layer overlays, these architectures can (a) better take into account network topology to ensure efficient data flow; (b) allow for multi-homed devices (for example a phone on both Wi-Fi and 3G networks); (c) certify the authenticity of content and (d) simplify the many overlapping and complex application protocols that give similar functionality.2

ICN proposals such as NetInf (Network of Information),3 PURSUIT,4 Named Data Networks (NDN),5 and Data-Oriented Network Architecture (DONA)6 establish a network-layer infrastructure that allows for the transfer of named objects. But these proposals do not thoroughly explore their architectures’ usefulness for dynamic applications. Although dynamic applications does not account for much Internet bandwidth, they account for much of the value of the Internet. Like static data, dynamic content can also benefit from ICN architectures’ scalability, redundancy, simplicity, and efficiency advantages.

As such, this paper will first examine how the four ICNs mentioned above can be used and adapted for dynamic programs. Since peer-to-peer networks are similar to ICNs, the paper then describes my attempt to build an environment for dynamic applications over BitTorrent, a popular peer-to-peer protocol.

ICN requirements

For an ICN to accommodate dynamic programs, it must first route to a copy of the program. The network then transports the input from the requester to the server. Finally, the network must transport the output back to the requester. These functions alone would be enough to build pretty much any application on it.

---

3 Dannewitz et al. 2013.
However, an ICN can provide extra functionality that increases its usefulness. It can route not only to a nearby server, but rather to the nearby server with the fastest resources. It can provide verification of program outputs. It can allow for execution by untrusted volunteers and automatically reconcile the data. It can route to servers that contain not only a program, but also specific data. These features in combination allow extremely powerful and flexible networks to be built.

Sample applications

Before going into the machinery of ICNs that can serve dynamic content, it’s important to have some context. Here are some examples to illustrate the power of ICNs that can also execute programs.

Layered distributed computing: web front-ends and MapReduces

Most large websites today are huge applications. The code looks to databases for user-data and content, rearranges that content according to the user’s input, and then puts it on a page to be displayed. However, since each application front-end is limited in processing power, load-balancing and CDNs are necessary.

A full-fledged ICN can take care of both of these problems. Each front-line server runs a single program with the same name; the ICN intelligently routes to an active front-line server, for example a Squid Cache. The front-line servers query servers in the next layer (for example, a set of Apache Web Servers) for data. The servers in this layer and all following layers can have unified program names as well. This process repeats recursively, up to the point of even the database front-end. If the ICN supports dynamic content, the ICN can manage load-balancing and locality to ensure low latency.

Even if there is no global ICN, companies can implement an internal ICN for inter-server communication within the internal network. The internal ICN can allow extra processing nodes to be put online dynamically to cope with demand fluctuations. Once the internal node is connected to the network, it can then publish its programs for execution within seconds. As such, the network is also seamlessly scalable.

MapReduce jobs are similar to large web applications: they are layers of inputs and outputs. A sample workflow involves preloading the data and programs, and then calling the top-level reduce program. A detailed workflow provides a specific example.

A master server preloads the data and programs

1. Each working server must be preloaded with a “downloader” program on all of the servers: the data server tries to keep it as close to the
2. Using the installed downloaders, the master pushes chunks of the input data onto the servers.
3. Since the executable programs themselves are usually fairly small, the master pushes all of the executables onto all of the servers.
Mappers and reducers run the program

- **Outputting data to the requester**: Each “map” or “reduce” task can either stream its output back in the response or merely send back a new named content identifier for the packed-up output. If it sends back a file identifier, the intermediate results can be cached for a while in the network if routers allow for it. For this example, we assume that each program the data back to the requester.

- **Making new sub-requests**: each task that requires a lower-level set of outputs then can make requests for that layer through the same ICN. If the ICN implementation supports it, the ICN can make a bundled request that routes to server with both a program and a required data file (for more details, see “Colocating data and programs” in the next section).

- **Resource management**: individual servers that are busy with computation tasks can either “unpublish” the content they have or communicate that in some other way to the network. Once they complete their task, they can publish the content again.

**Sweetening plain data: video transcoders and image resizers**

The first example shows that ICNs can actually perform an extremely diverse set of computational tasks and replace complex architectures. However, programs in ICNs can also act as sweeteners for simple pieces that live on servers that still primarily serve static data. This genre includes applications such as video transcoders and image resizers.

Static image files on large sites like Facebook, Yahoo, and Wikipedia are often shifted off the main domain to a Content Distribution Network (CDN). These CDNs provide caching and replication.

The CDNs can additionally perform more advanced tasks like built-in image resizing. Instead of the client, the server can resize the image and send it back to the user. This avoids wasting the user’s bandwidth and minimizes load times. In the case of videos, this operation becomes even more important as a too-large video may have too high of a bitrate for the user’s Internet connection: this would cause the video to stutter and fail to stream.

For an ICN to replace these more advanced CDNs, the ICN must implement more than just storage and retrieval of plain data.

- **Colocating data and programs**: the server can host both programs and data in a single package with the same name. A client wants a video “Video.mp4” but at a 480p resolution instead of its original 1080p resolution. The server can have a combined package of “Video.mp4; Transcode.exe” in a tar file and publishes it with the name “abcd.” The client can request that the server execute “abcd” with “480p” as an input, and the server will transcode the video contained in the package.

  - **Virtual packages**: if a server has a million videos, it can create a million archives, each containing Transcode.exe. This in itself is a waste. But what if the video transcoder executable got updated? It’s an even greater waste to have to repackage every single video file. Servers can instead implement virtual packages.
It publishes a package of “Video.mp4; Transcode.exe,” but instead of putting them into a real archive file, the server maintains a mapping between the published name and a virtual package. This is not a network feature, but is rather a local implementation feature. As such, it will not be discussed in detail in the following sections.

- **Compound requests:** alternatively, since the routing system of an ICN is aware of what data is where, a requester can submit a compound request if the ICN implementation supports it. For example, if a client requests “execute Transcode.exe with data Video.mp4,” the routing system can forward the request only to nodes where both of the named contents exist.

With executable programs in ICNs, videos could be transcoded on-demand. The vast majority of Youtube videos never receive more than a handful of views, so ICNs can produce space and bandwidth savings over just serving plain data.

**An analysis of basic features for dynamic content**

*Finding and routing applications*

Clients access data on Information-Centric Networks by specifying a unique identifier for the data. The routers take care of the rest. Program sources and binaries can be treated as just normal data, except we ask the server to execute the data rather than transfer it. As such, all of the ICNs examined can find dynamic programs. In addition, the locality and topology awareness benefits of data ICNs transfer automatically to dynamic programs.

However, filtering and routing may want to take into account systems’ advertised capabilities. For example, PUBLISH messages can be extended to advertise a rough measure of the computer’s processing power, perhaps using a standardized benchmark. The routing system can then take into account not only network distance, but also computational power when performing routing.

*Basic input-output*

Dynamic pages and applications available on the Internet are merely programs that take inputs from the user and produce outputs. For example, HTTP programs take inputs through the request header (cookies and URL parameters) and request body (POST parameters), and returns an HTTP header and HTML body in the response string. The applications may also store state, but they can do that without using the network.

ICNs usually do not function as two-way pipelines (like TCP) but as request-response transaction routers. The ICNs also control the routing. To ensure that all of the input reaches the same host and don’t end up fragmented, the protocol must have a way to include the input data in the retrieval request. However, we see that most ICN proposals can perform basic input-output with minimal modifications.
Detailed evaluation of architectures

NetInf: embed data in GET requests/separate transport protocol

The NetInf proposal is essentially a publish-subscribe protocol. It does mention dynamic data in its naming scheme: dynamic content is named using a hash of a public key that can be used to verify the content (the public key is included in the response). The conceptual protocol includes three message types: GET, PUBLISH, and SEARCH. The GET message requests a named-data object.

The proposal doesn’t give too many details about the GET message, but to provide input, the GET message should support additional data being included in it and maybe a flag to mark it as an execute request. The GET message must also be split if the input is large. Routers will probably need to use sequence and session identifiers to ensure that all of the GET messages reach the same host and aren’t split up and routed to random hosts.

The NetInf proposal gives an alternative where NetInf only implements a basic, integrity-protected message delivery service: a receiver initiates a transport connection on receiving a GET request. This would make two-way data transfer extremely simple.

Pursuit: embed data in PUBLISH requests

Pursuit/PSIRP uses a publish-subscribe protocol that’s similar to NetInf. It has a real SUBSCRIBE message in the place of GET. To request dynamic content, SUBSCRIBE would need to add the same features mentioned in the NetInf analysis.

DONA: separate transport protocol

The Data-Oriented Network Architecture (DONA) proposal also relies on hashed public keys for naming, but its protocol is only used to locate data: this makes two-way data transfer easy once again. Further data transfer runs over the traditional TCP/IP protocols. Applications simply first find the dynamic program (using hashes) on DONA and then establish a TCP connection to transfer data back and forth.

NDN

Named-Data Networking (NDN) is the worst-suited to dynamic content. Named-Data Networking (NDN) uses hashes with a hierarchal naming scheme for easier routing. But it does not use the pub-sub design of DONA and NetInf. Rather, requesters send INTEREST messages and routers forward them to multiple potential sources of the data. While the network can handle the duplication and forwarding of small INTEREST packets, larger packets can break the

---

7 Dannewitz et al. 2013.
8 Dannewitz et al. 2013.
10 Koponen et al. 2007.
network. When a router receives large *INTEREST* messages with arbitrary input and has to forward them toward different sources, the amount of bandwidth used by a single message can quickly multiply. To accommodate dynamic programs, NDN should be modified to guarantee that *INTEREST* messages be forwarded only along a single path. This requires fairly radical changes to NDN, and will probably force NDN to abandon its flooding-based method of finding information.

Because Named-Data Networking cannot deal with dynamic content easily without large changes, the rest of the paper will not address it in detail.

**Summary**

To serve dynamic content, the protocol needs to allow for two-way data transfer between the requester and a single host that has the requested program.

DONA actually uses TCP/IP after locating the data, and thus works with dynamic programs out of the box. NetInf and PURSUIT must allow payloads on their request messages. NDN’s use of request duplication can cause issues with large requests, and must be changed significantly to accommodate input and output.

**Advanced ICN features**

The four studied ICNs mostly have the essentials for application execution. Application execution on ICNs can be more efficient and effective with active resource management, program output certification, untrusted peer execution, and combined multiple-name queries.

**Resource management and overloading prevention**

Most websites with dynamic servers have complex load-balancing systems to ensure that content remains available with millions of active connections. Load-balancing is important for both static and dynamic content, but it’s especially important for dynamic-content servers since dynamic program execution scales worse than simple disk or memory access. On static-content servers, overloading usually reduces the throughput-per-user. On dynamic-content servers, overloading occurs with far fewer users. In addition, overloading usually reduces throughput more rapidly as more different types of resources – each of which can be a bottleneck – are shared between multiple context that have to switch between each other. Indeed, an 8-core video server reaches optimal load with just 8 requests!

There are a few ways that routers and servers can work together to load-balance.

- **Router load-balancing**: The routing system can act independently to avoid overloading. Routers close to the targets of the requests can keep statistics about the servers they’ve forwarded the request to. Upon receiving a new request, the router finds a server that has received relatively few recent requests. This technique requires no new network activity but places a high processing load on routers. In addition, it cannot take into account different workloads and server capacities. 5 requests on a Xeon MP server does not equal
5 requests on a Pentium II, and 5 requests to transcode a 3-minute video does not equal 5 requests to transcode a video.

- **NetInf (acting only as a message protocol) and DONA**: these two protocols can only track how many requests have been forwarded, and are worse at load-balancing. Note that the NetInf proposal specifically mentions that the routers can be configured to cache content by maintaining popularity statistics; this system can be grafted onto that easily.\(^{12}\)

- **NetInf (acting as a full protocol), PURSUIT, and NDN**: these protocols transfer data back through their protocols and thus routers keep track of state.\(^{13}\) As such, they have an exact count of how many executable are running (and not only how many execute requests have been forwarded). They are far more effective at taking into account differing job sizes and outputs.

- **Host load-balancing**: The serving system can choose to reject requests when it finds that it’s overloaded. When the routing system receives the retraction, it backtracks and then retries the request with another host with the data or program required. This allows for perfect overloading protection, as only the local server is contacted. However, when load exceeds a certain threshold, the routing system may need to backtrack through many extra round-trips, increasing latency.
  - **NetInf**: NetInf seems to have an assumption that limits the number of backtracks to one: only one router can own the object on each NetInf network (there can be multiple totally independent NetInf networks though: for example, one for the local subnet and one for the Internet).\(^{14}\)
  - **DONA, PURSUIT and NDN**: in PURSUIT each named piece can belong to multiple “scopes”, each of which has its own routing system. Similarly, every request in NDN and DONA can have multiple entries in the forwarding table at every router. The number of backtracking steps can be very large, and thus host load-balancing isn’t very practical in PURSUIT, DONA or NDN if the chance of overloading is high.

- **Distributed host load-balancing**: One solution to the long backtracking paths is to make each server route for their own content. In PURSUIT and DONA, a publish request by another node for the same named content can be passed down to every other node with the same named content. As such, every node in the network is aware of every other node that has the content. An overloaded node can then simply proxy the requests instead of executing them, leading to at most one-step backtracking again. This refinement does not help NDN, which doesn’t have publish messages.

- **Hybrid announce load-balancing**: A server can announce to all of the incoming routers when it’s overloaded. This tells the routers to remove or disable all of the routing entries

---

\(^{12}\) Ahlgren et al. 2012.

\(^{13}\) Ahlgren et al. 2012.

\(^{14}\) Dannewitz et al. 2013.
to it. When the overload stops, a new message can be sent telling the routers to add or re-enable all of the routing entries. This solution is somewhat traffic intensive, and requires the most work on the routers’ side. Overall, this solution isn’t very practical for any of the proposed ICNs, as the data is too centralized, as the “unpublish” request will have to propagate through the network, which takes a ton of time and packets.

For single-end-router protocols like NetInf and DONA, host load-balancing works best. For protocols that handle transport as well like PURSUIT and NDN, router load-balancing works best.

Program output verification

Among the four networks, DONA and NetInf both have self-certifying content. This means that the user can verify that the content returned is exactly what he requested. This usually involves hashing. Self-certifying content is essential for program execution.

The NetInf proposal was the only one to explicitly address the problem of naming dynamic content.\(^\text{15}\) For static data, NetInf uses a hash of the data as the name for the content. Checking the content is as simple as hashing the data. For dynamic data, NetInf proposes using a hashed public key as the data. Indeed, NetInf isn’t the first to come up with this idea. DONA uses a hashed public key as its “authority” part of its “authority:label” pair.\(^\text{16}\) The server then returns the public key in the answer to the request and uses the corresponding (probably non-public) private key to create a signature for the data returned.

To ensure the integrity of the resulting output, the client can just do the following:

1. Hash the public key the server sends in its response and verify that it’s valid by checking that it’s the request name.
2. Use the public key to verify the signature of the content.

While DONA and NetInf both describe client-side verification, this scheme can be extended to have the routing server do a similar check to reduce the potential for denial-of-service attacks. If only the host verifies the validity of servers’ outputs, a large number of malicious servers can overwhelm a single good server. DONA presents a way for a valid and non-malicious routing system can present a test to prevent hosts from getting onto the network without the proper key in the first place.\(^\text{17}\)

1. The server first receives a publish message for content with name \textit{content\_name} from a server.
2. It sends the server a sample nonce.

\(\text{15}\) Dannewitz et al. 2013.
\(\text{16}\) Koponen et al. 2007.
\(\text{17}\) Koponen et al. 2007.
3. The server uses the private key corresponding to the public key that hashes to 
   content_name to sign the nonce, and then sends both the public key and the signature 
   back.
4. The router then only adds the server to the routing table once it has
   a. verified that the public key hashes to the name and
   b. verified that the nonce was signed by the corresponding private key.

This simple procedure can actually be used on PURSUIT by applications if they wanted to. PURSUIT content are published with a rendezvous identifier (Rid) that can be “derived by an application-specific function.” Thus the PURSUIT application can simply choose to use the hash of the public key as the output of its application-specific function.

This process is essential to maintain trust. In traditional host-based processing, we know that hosts are owned by specific parties who can guarantee the correctness of results. However, anyone can claim that their program has correct output. The solution is to only distribute private keys to those execution systems that you trust.

**Untrusted peer execution**

Let’s say we have a program that we want volunteers to run for us. This would be a peer-to-peer execution environment like BitTorrent. With static files, we could verify data sent by untrusted people simply by hashing it, but we can’t do that with executables.

If the executables are run on clients that aren’t controlled and trusted by a central authority, there is one fairly reliable way to verify the output that’s currently used by distributed computing projects such as BOINC. It’s simply to execute the job on multiple clients. While we can never be entirely sure that the output is correct and safe, popular content with a lot of legitimate volunteering users will tend to corroborate each other.

To add this functionality, we need to add a field in the GET, SUBSCRIBE, Find, or INTEREST message of the corresponding ICN implementation called: “redundancy.” If we set the redundancy to 3, the routing system multicasts the message to three hosts. The routing system should try to ensure that the hosts are on different sub-networks to prevent a cluster of malicious computers from supporting each other. To do this, routing system should look to split the request as soon as possible (as close to the requester as possible).

Servers may also choose to reject requests with more than a certain number of “redundancy” to discourage wasteful requests and to prevent one request from tying up a lot of resources and thus denying service to others.

This functionality can be implemented on all four of the analyzed systems. Each of them already have multiple routing paths per content to split on. Again, NDN is a bit harder to deal

---

18 Fotiou et al. 2012
with, since it usually floods the network to look for copies first.\textsuperscript{20} It will come back with a few copies, of which the requester will usually request one.

**Multiple-name content queries**

In the video transcoding example mentioned earlier, I wrote that it would be better if a request could be made to search for a server with two or more pieces of content on it: for example, a client may use such a request if he wanted to execute “Transcode.exe” on only a server also with “Video1.mp4” already on it.

If every routing server contained a full routing table that mapped each piece of content to every end server that had it, this would be easy. We just lookup all the pieces and then intersect the hosts that turn up from each request. We then choose the closest of the surviving hosts.

However, most of the ICN implementations do not keep full end-host routing tables. For example, NetInf maps content names to the next-hop router only to save on resources.\textsuperscript{21} That’s still fine: assuming that content on the same end-server will have the same paths, the content will still eventually be found.

This puts a slightly greater demand on routing servers: instead of just doing a search through a lookup table for a single entry, routing servers would need to do intersections, which involves scans of lookup tables or extra data structures. While multiple-name content queries may be useful, their primary use-case of finding data and programs together can be replaced with more elegant solutions, such as the Virtual package described in the Examples section.

**An ideal ICN for execution**

An ideal ICN for execution would consist of the following features. These features were chosen because they’re simple and work well in combination.

1. **NetInf-like naming – a data (for static content) or public key (for dynamic programs) hash.** This allows for verified ownership. The owner of the program can distribute the program (with the private key inside) only to the systems they want to run it on by marking the content as executable-only. Alternatively, the owner can allow it to be distributed freely as a downloadable file, where any peer can download it and run the executable.

   a. **Program output verification** is included with this naming scheme.

2. **DONA-like publish/find messages.** These messages are the bare essentials for routing and keep the system simple. Extra functionality like “search” can be implemented in application layers.

3. **A separate host-based transport system: TCP/IP.** Once the server hosts have been found, all input and output must be between those two specific hosts for execution to


\textsuperscript{21} Dannewitz et al. 2013.
work. Using a TCP/IP transport puts us on the solid ground of an established protocol and makes it easier to port existing applications.

4. **Distributed host load-balancing:** Since retries after rejects only require one hop back (to the initially-chosen host), routers can retry with fairly low overhead.

5. **Untrusted peer execution support:** This involves just an extension to the “find” message: instead of returning a single host, they can return multiple hosts.

This combination allows for self-verifying, load-balanced execution of replicated programs, with a mature transport protocol handling the actual data-transfer.

It is impossible for this network to verify outputs from untrusted peers, as that output is sent over another transport protocol which we cannot observe. Verification can still be implemented at the application-level.

**From Information-Centric Networks to BitTorrent**

Many of the main advantages touted ICN proposals have been worked on outside of the network layers. Whereas ICN proposals focus on bringing features of current application protocols down into the network level, Application Level Traffic Optimization (ALTO) brings the awareness of network topology up into the application level. Researchers have already proposed a topology-aware version of BitTorrent.\(^\text{22}\) BitTorrent’s peer-to-peer structure – with its separate routers and end-users – is a good candidate for testing out ICN concepts.

**BitTorrent basics**

BitTorrent is a peer-to-peer file transfer protocol. To use BitTorrent:

1. The client opens a .torrent metadata file in a BitTorrent client. This .torrent file usually provides the client with the unique identifier for a package of data, as well as the hostnames of trackers (centralized server that keeps a table mapping content to hosts) and hashes of all of the pieces.

2. The client (one of many “peers”) announce themselves upon startup to one or more trackers or to a distributed hash table (DHT).\(^\text{23}\)

3. The tracker sends clients a list of all of the other “peers” who’ve announced themselves in the last while (usually in the last half hour).

4. The BitTorrent client then can manually initiate connections to any of the many peers.

5. When connected to each other, the two peers exchange information about what pieces of a file each other has, and tells each other whether they’re interested in each other’s content. If both peers are uninterested, the connection is closed.

---


6. If a peer has too many connections already or would rather upload to someone else, it sends a “choke” message to the other peer. All connections start out choked.

7. If a peer wants to upload again, he “unchokes” the other peer.

8. The “unchoked” peer can then make a request for a piece.

9. The requesting peer receives the piece and verifies it by hashing it and checking it against the hash provided in the .torrent file.

BitTorrent is already quite similar to the “ideal ICN” I named above. (1) Static content is identified by an info-hash: a hash of the names of the files. (2) Publish/find messages are exchanged in the form of “announces” to trackers. Trackers act as flat routers. (3) Actual file transfer is performed over a peer protocol built on top of TCP/IP. (4) Hosts manage their own load-balancing. On failure, clients simply connect to another client. BitTorrent has an additional advantage in that each Torrent is its own mini-network: as such, network paths are small, and each peer can usually hold a local database with all other peers.

**Extending BitTorrent to allow execution**

The infrastructure for executing BitTorrent involves adding a new set of “extension messages” using the libtorrent-rasterbar extension interface. The new extension is called “execute.” When connecting to a new client, the handshake will indicate to peers that this extension is supported.

There are two implemented message types: “request_execute” and “full_response.”

The request_execute message contains a request_id for tracking the request, as well as a body that contains the initial input. The client peer (that wants to execute the program) sends this message to the program on the server peer.

The server peer receives the request_execute message. It first verifies that it has a completed version of the executable package – a directory. The server peer then looks inside the /MANIFEST.torex file of the package to find the command-line argument to execute. It executes this command from that same directory using the runtime environment. The server peer waits for the execution to complete, reads all of the output from STDOUT, and then sends it back in a full_response message.

The full_response message contains the same request_id that the server sent. It also contains a status code which can be used in case of errors (such as incomplete package, no binary found, and so on). It contains a body that includes all of the standard output.

**Further protocol extensions**

Further extensions should allow for streaming input. Instead of a single message of “request_execute,” there can be additional messages “request_continue” and “request_end” that allow the user to send more data to be fed into the server peer’s standard input for the program.

---

Similarly, a “partial_response” message can be used to indicate that more output may be coming.

Benchmarking of overhead

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect to tracker, get peers, connect to peer</td>
<td>3143</td>
</tr>
<tr>
<td>Send request</td>
<td>3</td>
</tr>
<tr>
<td>Run stub program</td>
<td>24</td>
</tr>
<tr>
<td>Send response</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 1: Connection initiation from scratch is slow, but the overhead of running programs is fairly low. Overall, this suggests that peers should try to pre-emptively connect to other peers if they can anticipate needing to use it. This should be the first area of focus for improving connections. The experiment was conducted over a local area network with <1ms RTT.*

Future work and issues

**Security**

If the BitTorrent client is going to run arbitrary data, it must use a sandbox to reduce the risks of data execution. The most viable candidates right now are LXC and the Google Chromium sandbox. The Chromium sandbox is the only one that provides free and decent protection on Windows, so that should be the go-to choice for any generic client.

**Resource management**

As described in the “Advanced ICN features” section above, resource management is a major issue. BitTorrent’s unchoke/choke method can be adapted easily to fit the resource management and load-balancing roles. When the server peer feels that it’s overloaded, it simply chokes all of the peers.

In addition, resources must also be shared properly between processes. Server priorities and affinities should ideally be controlled automatically to provide each client peer with the same amount of processing power.

**Faster connections**

Connecting to a tracker, getting peers, and connecting to peers is very slow. Granted, normal BitTorrent trackers are not optimized for quick connection, but this is the biggest performance bottleneck for more robust uses of Internet connections.

One implementation may involve:

1. Using UDP or long-lived TCP connections to avoid round-trips to the tracker.
2. Make peers have an in-memory GeoIP cache to determine which peer to connect to and quickly choose a close peer.
Conclusion

Information-Centric Networks have the potential to simplify the many layers that have accrued on top of the legacy Internet to provide the scalable and redundant services that we use today. The ICN proposals have paved the way for data to flow naturally through the Internet, without needing application-layer protocols to muddle it up. Bringing rich applications to ICNs allows the applications to take advantage of the clarity of ICNs as well. The analysis above has shown that with very few changes, the NetInf and DONA architectures can become platforms not only for the transport of data, but also for the transport of inputs and outputs to networked programs. This opens a whole new door.

On the other hand, BitTorrent has managed to implement many of the proposals of Information-Centric Networking in incremental steps. It contains trackers for routing and content verification techniques. It contains an extensible package that makes it easy to build on top of that routing system. It may be worth exploring BitTorrent as a well-supported, practical and incremental base for developing a pseudo-network layer that provides ICN functionality.
References


Appendix: Implementation Notes

The modified libtorrent application can be found at https://bitbucket.org/hsource/libtorrent-execute

To compile, clone it and make it. Simply run `./configure` and `make`. Note that boost is required.

To run a test client, go into /examples.

- To run a server-like node: do `./simple_client.py executable.torrent n`
- To run a client-like node: do `./simple_client.py executable.torrent y`

Wait a few seconds for the each client to connect. Enter a character in the terminal and press enter on each side. On the server-side node, that will just show you the statistics. On the client-side node, that will try to submit a request.