GRAND: Git Revisions As Named Data

Jan Janak, Jae Woo Lee, Henning Schulzrinne
Department of Computer Science, Columbia University, New York, USA
{janakj,jae,hgs}@cs.columbia.edu

ABSTRACT

GRAND is an experimental extension of Git, a distributed revision control system, which enables the synchronization of Git repositories over Content-Centric Networks (CCN). GRAND brings some of the benefits of CCN to Git, such as transparent caching, load balancing, and the ability to fetch objects by name rather than location. Our implementation is based on CCNx, a reference implementation of content router. The current prototype consists of two components: `git-daemon-ccnx` allows a node to publish its local Git repositories to CCNx Content Store; `git-remote-ccnx` implements CCNx transport on the client side. This adds CCN to the set of transport protocols supported by Git, alongside HTTP and SSH.

1. INTRODUCTION

Content Centric Networking (CCN) is a new network architecture that addresses some of the well-known shortcomings of IP networks [10]. In IP based networks the underlying network architecture deals with connections between hosts. This requires applications to know the location of the data it wishes to obtain. Complicated overlay systems, such as Content Delivery Networks (CDN) and Peer-to-Peer Networks (P2P), emerged as a result. CCN eliminates the need for such overlay networks. In CCN, applications fetch data by name rather than by location, leaving the specifics of storing and locating data to the network itself. Project CCNx [7] is a reference implementation of CCN node and application libraries.

The project offers developers a way to implement applications on top of CCN. CCNx-based applications benefit from transparent caching on nearby nodes, native support for broadcast and multicast data dissemination, and built-in data security. A community of researchers porting applications to CCNx has been building around the project. Such attempts are important to gain insight into whether CCN can offer any advantages over IP networks for these applications. In this paper we present our contribution: a port of Git to CCNx.

Git is a Revision Control System (RCS) for managing source code. Unlike traditional centralized systems, such SVN [8] or CVS [1], Git is a *distributed* system. Most operations in Git only access the local copy of the repository, as opposed to their counterparts in SVN or CVS which are performed over the network. Git only requires network connectivity to synchronize the local repository with a remote copy. To synchronize two repositories over the network, Git needs to transfer missing objects to each other. Git supports several protocols including HTTP, SSH, and the Git server protocol [4].

Configuring the infrastructure needed for network synchronization is not necessarily trivial. It requires setting up a web server, a Git server, or giving users shell access. Hosting sites like Github [5] can make that process easier, but in that case one has to rely on a centralized third-party service. There was a previous attempt to use the Bittorrent peer-to-peer network [6] to synchronize Git repositories, but the project has been abandoned.

Git is an appealing application for CCN. It's data model is a good fit for CCN's named data. Git's network synchronization can benefit from some of the advantages offered by CCN, such as automatic caching at nearby nodes, built-in security and broadcast data distribution.

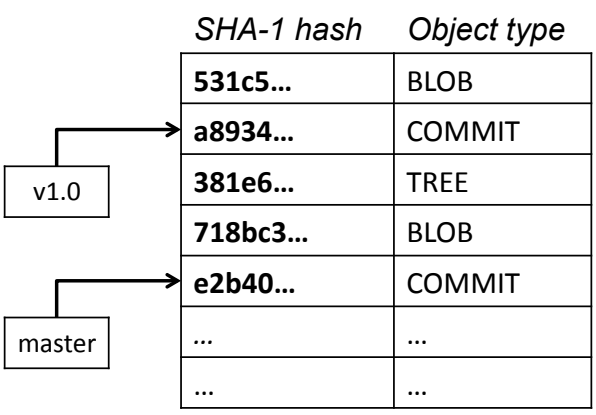
We extended Git by adding CCNx to the set of supported transport protocols. Our implementation can be used to synchronize Git repositories over CCNx network. A node can export a Git repository via CCN. Another node can clone the whole repository or fetch incremental updates.

The rest of the paper is organized as follows. In Section 2 we provide background information on Git internals and a brief overview of CCNx. Section 3 describes the architecture and implementation details. We conclude and discuss future work in Section 4.

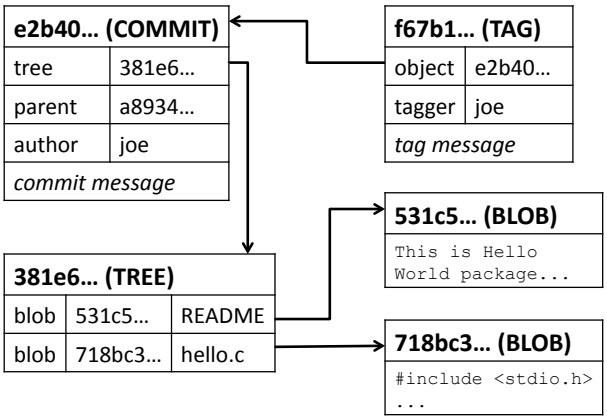
2. BACKGROUND

2.1 Git

Figure 1(a) shows the layout of a Git repository. The repository is a simple key-value database stored on the



(a) Git repository as a flat database of objects identified by their SHA-1 digests. Branches are pointers to individual objects.



(b) Object types and their relationship. A tag points to a commit; the commit references a filesystem tree; the tree binds filenames with blobs; blobs contain file contents.

Figure 1: Simplified model of a Git repository.

local filesystem. Objects are stored either in individual files, in compressed files called "packs", or in combination of both. Each object is immutable and uniquely identified by a SHA-1 digest of the object's data. Cryptographically generated IDs are necessary because Git works without a central authority. In addition to objects, a repository contains a set of pointers called *refs*. Refs point to objects using their IDs and they are typically used to implement branches, i.e., separate lines of development. Figure 1(a) shows two such refs: `master` points to the latest commit on the main branch; `v1.0` points to the latest commit on a stable branch.

There are four types of objects: Commit, Tree, Blob, and Tag. Figure 1(b) shows the relationship between them. When the user commits a set of changes into the repository, Git creates a new Commit object. The commit object contains author information, commit message, IDs of parent Commit objects, and an ID of a Tree object which represents the state of the top-level directory. A Tree object is a directory listing. Each entry in the list contains the ID of a Blob a file or another Tree object for a subdirectory. A Blob contains the content of a file. Tag objects are used to assign labels to Commit objects. The binding between the label and the Commit object can be signed with the user's private key.

Objects in a Git repository form a directed acyclic graph (DAG) through their pointers to other objects [2]. To synchronize two repositories, Git obtains the list of branches from both. For each pair of branches that differs in IDs, Git transfers any missing objects by traversing the DAGs. In order to support this operation, the transport protocol must implement two basic primitives: obtain a list of all branches and fetch an individual object by its ID.

2.2 CCNx

There are two types of packets in CCNx: *Interest* and *Data*. Applications send Interest packets and receive matching Data from the network. An Interest packet contains a name prefix, such as `/git/linux/refs`. Any Data packet whose name matches the prefix can satisfy the Interest packet. A Data packet can only be sent to the network if there was a matching Interest. This maintains a flow balance in the network.

When an Interest packet reaches a CCNx Content Router, the Content Router searches its local cache of Data packets, called the *Content Store*, for a matching Data packet. If a matching Data packet is found, the Content Router sends it back, satisfying the Interest packet. CCNx requires that each Data packet is cryptographically signed. This prevents the caching node from tampering with the Data packet.

If there is no matching Data packet in the Content Store, the Content Router saves the Interest and the *face* it came from in the Pending Interest Table (PIT). A face is a CCNx generalization of the network interface. The Content Router then forwards the Interest packet to one or more outgoing faces based on the Forwarding Information Base (FIB). Figure 2 contains a simplified diagram of the CCNx Content Router, labeled as "ccnd" in Node A.

When the Content Router receives a Data packet, the Content Router searching the PIT for a matching Interest. If a matching Interest is found, the Content Router forwards the Data packet over the face recorded in the PIT entry, optionally caching the Data packet in the Content Store. If there is no matching entry in the PIT the Content Router drops the Data packet. PIT entries in Content Routers record the path of an Interest

packet across the network and are used to forward the matching Data packet back to the source.

3. GIT OVER CCN

We extended Git with support for Content-Centric Networking based on CCNx. The extended version can export contents of Git repositories via CCN and synchronize local copies with remote repositories published via CCN. We focused primarily on scenarios using local area networks with multicast in our experiments due to lack of global routing in the current CCNx prototype.

3.1 Architecture

Figure 2 illustrates the overall architecture of the system. Each node willing to join the CCNx network must be running a local instance of ccnd, the CCNx-based Content Router. For our initial experiments we configured a small local area network and used multicast to disseminate Interests among all nodes in the network. Each node in the network can run both the client and the server part simultaneously if needed.

Nodes running the server daemon `git-daemon-ccnx`, which is described in Section 3.2.1, can export the contents of their local Git repositories to other nodes in the network. For example, users who have local clones of well-known repositories can make them available to other users on the same network via CCNx, saving bandwidth and costs associated with obtaining the contents of the repository from a remote server.

A node that wishes to synchronize its local copy with a remote repository published via CCNx runs the Git remote helper `git-remote-ccnx`, which is described in Section 3.2.2. The purpose of the remote helper is to translate requests for data to CCNx Interests and store the data obtained from the network in the local repository.

Our implementation follows a simple naming convention as shown in the top right box in Figure 2. Every CCNx Name begins with a prefix common for all Git repositories. We use `/git` as the common prefix in all our examples. The component which immediately follows the common prefix contains the name of the Git repository. The repository name should be preferably same as the name used by the official public repository of the project (without the optional `.git` suffix). This simple convention allows clients to identify multiple clones of the same repository available from CCN. The next Component identifies the type of data and can be either `refs` for lists of branches and tags or `objects` for individual objects from the repository. Reference lists have a timestamp-based version in Name. Individual Git objects are identified by their SHA-1 digest in ASCII form in Name. If an object was fragmented then the last component of Name contains the fragment number.

3.2 Implementation

Our implementation uses the client library from the CCNx project [7]. All the code is implemented in C on top of the latest Git source code available from its development repository.[1] Our application consists of two parts: A server daemon called `git-daemon-ccnx` and a Git remote helper called `git-remote-ccnx`. Figure 2 illustrates both components, their building blocks and their relationships to other components of the system. We describe both components in detail in the following sections.

3.2.1 Server Daemon

The server part is implemented as a standalone daemon running permanently and waiting for Interests from ccnd. The daemon is configured with a directory on the local filesystem which contains Git repositories to be exported over CCN. It connects to the local content router ccnd upon startup and registers a "filter" for Interests with the prefix common for all Git repositories in the system (`/git` in the figure). The registration creates a new entry in the Forwarding Information Base (FIB) of ccnd. The daemon maintains the connection and registration permanently for its lifetime. When ccnd receives an Interest that cannot be satisfied with existing data from the Content Store and whose prefix matches the prefix registered by our daemon, it forwards the Interest over to the daemon via face 5, an internal network interface created by ccnd when the daemon connected to it.

Upon receiving the Interest from ccnd, the daemon extracts the name of the Git repository from the Interest's prefix and checks whether the repository exists and is accessible. If it is, the daemon spawns a new worker process to handle all further Interests for that particular repository. Having one process per repository was necessary due to implementation peculiarities of Git. Being primarily a command line oriented tool, Git functions often terminate the current process on error. Handling requests for individual repositories in separate processes allows the daemon to continue running in such cases.

The new worker process opens a new connection to ccnd, which creates a new face, and registers a filter to match Interests for its repository. In addition to the common prefix, the filter would also include the name of the repository. That makes the corresponding FIB entry longer than the entry created by the dispatcher process. Because ccnd uses longest prefix match to select outgoing faces for Interests, it will prefer the worker's connection for Interests matching its repository name over the connection of the dispatcher process. That is, the dispatcher only receives Interests for repositories

[1]Git does not contain any shared libraries or header files which would make it possible to distribute our code separately without reimplementing significant portion of Git.

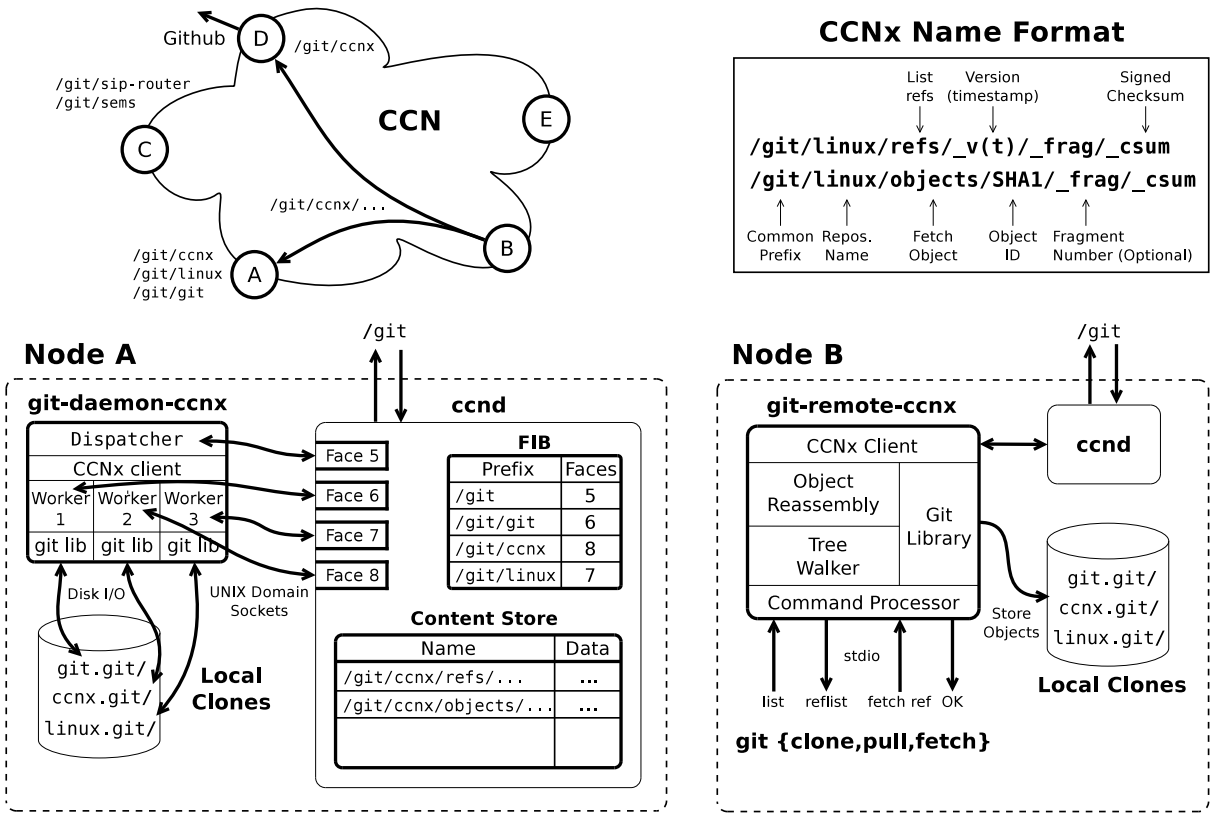


Figure 2: System architecture. Three repositories on Node A are exported by `git-daemon-ccnx` to the Content Store of local content router. Git on Node B uses `git-remote-ccnx` to translate requests for Git repository objects to CCNx Interests with Names following the scheme shown in upper-right box. The network diagram illustrates an exchange of packets among nodes in a small local network.

that have no dedicated worker processes running yet. The worker generates a Data packet from the repository to satisfy the Interest and sends the Data to ccnd. In addition to forwarding the Data upstream, ccnd may cache it in its Content Store. Our current implementation supports two types of Interests with prefixes shown in the Name Format box in Figure 2: requests for lists of references and requests for individual objects.

Upon receiving an Interest with `refs` component following the repository name in the prefix, the daemon generates a Data packet with a list of all branches and tags from the repository in a custom binary format. For each branch the list contains the SHA-1 of the latest commit on that branch; for each tag it contains the SHA-1 of the object the tag points at. The daemon also adds a timestamp-based version component to the Name of the Data packet. Clients can use that component to request the latest version of the list regardless of versions available from the Content Store. The daemon signs the content of the Data packet with a preconfigured private key, but our current client prototype does not verify the signature yet. Because branches may get updated often, the Freshness attribute of the Data is set to a low value (in the order of seconds). Clients use such lists to determine which objects they need to fetch in order to synchronize their local copy of the repository.

To obtain an individual object, the client sends an Interest with `objects` component followed by the SHA-1 ID of the desired object. Upon receiving the Interest, the daemon obtains the object data from the corresponding Git repository, encapsulates the object (in its compressed form) in a Data packet and sends the Data packet to ccnd to satisfy the Interest. If the daemon couldn't find the desired object in the repository, it does not respond to the Interest. A failure to locate the object indicates that the local repository may be incomplete, rather than invalid or corrupted. A daemon running on another node may have a more recent copy of the repository and could satisfy the Interest. The Freshness attribute in Data packets carrying Git objects can be set to a high value because Git objects are immutable. Once a copy of the object is in the Content Store, there is no need to refresh it.

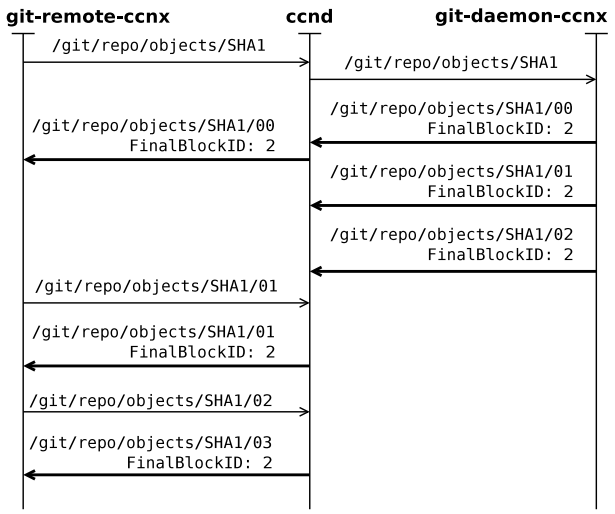


Figure 3: An example exchange of Interest and Data packets to retrieve a Git object that needs to be fragmented. The client generates additional Interests to retrieve missing fragments from Content Store.

Objects stored in Git repositories can be large and may not fit into a single Data packet. In that case the daemon has to split the object into a set of fragments. Each fragment is then encoded into a Data packet of its own. To facilitate reassembly in the client, such Data packets must contain additional information. The daemon appends the number of the fragment (counted from 0) as the last component of Data's Name. The FinalBlockID attribute in ServerInfo section inside the Data packet is then set to the number of the last fragment in the set. All fragments are then uploaded to the Content Store, one after another. Figure 3 illustrates that scenario.

It may also happen that the daemon receives an Interest for an individual fragment (having the number of the fragment in the last prefix component). That can happen, for example, if one of the fragments expired or was removed from the Content Store. In such case our current implementation repeats the steps described before and simply re-uploads all fragments into the Content Store in CCN.[2]

3.2.2 Remote Helper

The client part, labeled as Node B in Figure 2, is implemented as a *remote helper*. Remote helpers are standalone programs used by Git to access remote repositories over a variety of transport protocols. Our remote helper implementation for CCNx is called `git-remote-ccnx`. Git uses the remote helper whenever it needs to interact with a remote repository available over CCNx. Such repositories are identified with URLs starting with `ccnx` scheme. When the user runs `git-clone` or `git-fetch` to obtain the latest objects from such a repository, Git starts `git-remote-ccnx` remote helper, connecting to its standard input and output via UNIX pipes and setting the local copy directory to be modified as the current working directory.

The remote helper receives commands from Git over its standard input and translates them into CCNx Interests. Each Interest will have a prefix constructed from the URL of the remote repository:

```
[remote "origin"]
  fetch = +refs/heads/*:/refs/remotes/origin/*
  url = ccnx:/git/ccnx
```

All Interests for data from repository "origin" in the configuration example above will have prefixes starting with `/git/ccnx`.[3] Upon receiving Data packets the remote helper either sends their content to Git over its standard output (for branch lists) or writes the data into the local repository (for individual objects). Our current remote helper implementation supports two commands: `list` and `fetch`.

To obtain the list of all branches and tags available in a remote repository, Git sends the `list` command to the remote helper. The Command Processor parses the command, constructs a CCNx Interest and passes it to the local instance of ccnd. When the corresponding Data has been received, the Command Processor extract the list from the packet and compares the state of remote branches and tags with those in the local repository. For any branches and tags that differ in their SHA-1 value, Git sends a `fetch` command to the remote helper, asking the remote helper to fetch any missing objects into the local copy of the repository.

The Tree Walker component[4] of the remote helper starts fetching missing objects, one by one, following references and links in Git objects as shown in Figure 1(b), until all missing objects have been obtained from the remote repository. For each missing object it creates the corresponding Interest and waits for matching Data to arrive. The content of the Data packet is then stored in the local repository. If the received Data contains only a fragment of the requested object, the Tree Walker passes it to Object Reassembly.

Given an object fragment, the Object Reassembly component generates a series of Interests to fetch all missing fragments needed to complete the object. Be-

[2]Uploading multiple Data packets in response to one Interest should be fine in this case, because ccnd is running on the same host as our daemon and the Data packets will not be forwarded any further.

[3]To teach Git how to recognize URLs with only one leading / we had to improve its built-in URL parser.

[4]Our implementation re-uses the tree walker implementation from HTTP remote helper.

cause each Data fragment carries its own fragment number as well as the number of the last fragment, it does not matter which fragment from the set was received first, each fragment contains all the information needed to obtain the rest. After all fragments of the object have been downloaded, the Tree Walker writes the complete object into the local repository and continues traversing the object tree.

3.3 Security Considerations

Individual objects in the Git database are identified by their SHA-1 digest calculated over the content of the object. Git verifies digest for every object retrieved from the local database and reports an error if it is invalid. This catches corrupted and maliciously modified objects. To protect the integrity of a repository snapshot, the developer can create a tag and sign the tag object with their private key. This allows other developers to verify the integrity of their repository copy because they can verify the signature if they have the public key. This mechanism allows us to verify the integrity of Git data obtained from CCN although our current prototype does not implement CCNx-based security yet.

4. CONCLUSION & FUTURE WORK

We extended Git by adding CCNx to the set of supported transport protocols. Our implementation can be used to synchronize Git repositories over CCNx network. A node can export a Git repository via CCN. Another node can clone the whole repository or fetch incremental updates. We have successfully cloned some of the biggest repositories, including the Linux kernel. Our implementation is freely available from [3]. We also setup a publicly available CCNx node with a number of Git repositories to fetch from to encourage the CCNx community to experiment with Git over CCN. The web page contains detailed instructions on how to clone one of those repositories.

Our implementation is in early stage and has several shortcomings. We only implemented the most basic mode of network synchronization where objects are transferred one-by-one, rather than in packs. This makes our implementation slower than other existing transport protocols. We plan to implement more efficient synchronization in the future. Our client currently does not verify the integrity of Data objects obtained from the network. We rely on Git security described in Section 3.3. Integrating CCNx security remains as future work. Lastly, we plan to develop a better naming scheme for Git objects. That would allow us to support more sophisticated collaboration scenarios that are difficult to achieve using existing transport protocols, such as discovery of repositories, or automatic caching of Git repositories for backup and load balancing.

Our implementation can already be used to perform actual development work with Git over CCN. Both cloning and fetching of remote repositories are supported, allowing the user to create the initial clone, as well as to fetch incremental updates from the network. Because the CCNx project manages its source files in Git, our extension makes it possible to use Content-Centric Networking to develop CCNx itself!

5. REFERENCES

[1] Cvs: Concurrent versions system. `http://www.nongnu.org/cvs`, Feb. 2011.

[2] Git for computer scientists. `http://eagain.net/articles/git-for-computer-scientists/`, Feb. 2011.

[3] Git over content-centric networks. `http://janakj.org/git-ccn`, Feb. 2011.

[4] Git server protocol. `http://git-scm.com/gitserver.txt`, Feb. 2011.

[5] Github. `http://github.com`, Feb. 2011.

[6] Gittorrent. `http://code.google.com/p/gittorrent/`, Feb. 2011.

[7] Project CCNx. `http://www.ccnx.org`, Feb. 2011.

[8] Subversion. `http://subversion.apache.org`, Feb. 2011.

[9] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard. VoCCN: voice-over content-centric networks. In *ReArch '09*, pages 1–6, New York, NY, USA, 2009. ACM.

[10] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CoNEXT '09*, pages 1–12, New York, NY, USA, 2009. ACM.

[11] M. Meisel, V. Pappas, and L. Zhang. Ad hoc networking via named data. In *MobiArch '10*, pages 3–8, New York, NY, USA, 2010. ACM.

[12] J. Saltzer. On the Naming and Binding of Network Destinations. RFC 1498 (Informational), Aug. 1993.

[13] D. Smetters and V. Jacobson. Securing network content. Technical report.