PRELIMINARY PROJECT PROPOSAL

Redundancy-Aware Peer-to-Peer Protocol (RAPP)

Bernhard Ömer

20th October 2000

Abstract

This proposal is about developing a distributed peer-to-peer file-sharing system which is able to automatically identify multiple instances of identical data and takes advantage of this redundancy.

The main research effort is to investigate, how the above concept can be used to improve scalability, efficiency, robustness, load-balancing and anonymity in peer-to-peer file-sharing systems and the development of a protocol (RAPP) together with a reference implementation which allows the construction of a content-addressed, searchable file-sharing network on top of the existing IP-based infrastructure of the Internet.

E-mail: `oemer@tph.tuwien.ac.at`
Homepage: `http://tph.tuwien.ac.at/~oemer`

1 Motivation

1.1 Client-Server Systems

While the Internet itself has been designed as a decentralized peer-to-peer network, most IP-based information services and protocols imply a client-server architecture. This is especially true for file-transfer protocols like ftp, http or nfs. With the rising numbers of users and the ever increasing availability and popularity of storage-intensive multimedia content, the principal limitation of client-server concepts becomes more and more apparent: As bandwidth requirements for client-server architectures scale with the size of the content as well as with the number of users, the reliable distribution of even moderately sized content causes considerable costs on the side of the publisher.

While this problem can in principle be addressed by the use of mirror and proxy servers, both approaches require a considerable amount of administration overhead on the side of the publisher and good will on the side of the consumer.

1.2 Peer-to-Peer File-sharing

One might argue that the above problem is inherent to the process of distributing content, but this is not the case: Usually, the combined resources of all parties involved in the process of delivery would be more than sufficient to allow for an efficient distribution of even very large files, if all, or at least some of them would chose to offer downloaded content again for redistribution and – equally important – let the others know about it before they chose to consult the original publisher.

Peer-to-Peer file-sharing systems (FSSs) try to at least partially automate this process, by

- enabling each user to act as a server as well as a client,
- providing automated means to announce and search content and
- defining a protocol which allows different users to communicate this content.

Many common FSSs like e.g. napster or scour.net only decentralize the actual data transfer and delegate the the arrangement of connections and all search functionality to a single directory server.

Real peer-to-peer systems like gnutella or freenet avoid the use of centralized servers (and thus eliminate any single point of failure) by organizing all users into a loosely coupled network of equal nodes.

1.3 Specifying Content

Digital content usually comes in the form of binary files. The only means that most operating systems provide to specify the content of a file, is by an unstructured textual name which, at the same time, severs as content description and (in combination with the pathname) as access key to the filesystem. Consequently, most FSSs address files as a tuples of node-ID, pathname and filename (just like HTTP-URLs).

Since filenames are basically arbitrary, they are in an n-to-m relation to the actual data. This has two serious consequences which limit the the above aspect of FSSs as implicit mirroring/proxy-networks:

1. **Integrity:** A user can never be sure whether two files of the same name actually contain the same data.
2. **Polymorphism:** A user also can't be sure whether two files with different names are in fact different.

This leads to several practical problems, e.g.

- A user never knows how many distinct files match his search criteria.
- Downloads have necessarily to be restricted to point-to-point connections.
- Load-balancing through simultaneous partial downloads from multiple hosts are impossible.
- Partially downloaded files have to be discarded if the peer leaves the network.

Current FFSs basically use two ways to address the above issue. Systems like freenet rely on user-provided, descriptive keys and provide means to ensure that any key is globally unique within the system. However, besides the administration overhead for providing the descriptions, this only solves the integrity issue.

Another possibility, which doesn't require user interaction, is to tag files with hash values. At first glance, one might think that this would solve both problems; this, however, is not the case: Since in a scalable distributed system no node can be expected to know about all files within the system and only the filename-space is searchable, using hash-values as tags can only eliminate the polymorphism problem within the results of a single query.

2 Redundancy-Aware Peer-to-Peer Protocol

2.1 Content Based Addressing

The above problems are rooted in the fact that conventional FFSs only maintain one searchable database of filenames plus their location within the network and consequently suffer from the same n-to-m relation from filename to file content as conventional filesystems

The main concept of redundancy-aware FSSs is to separate the search process into two parts: The search for filenames and the search for instances of a certain file. This can be realized by introducing a hash-value of the content as a global file identifier (GFID).

The system consequently has to maintain two global, searchable databases: A name-database (NDB) which holds all known names for each GFID (n-to-1) and a host-database (HDB) with, for each GFID, maintains a list of all nodes hosting an instance of the file (1-to-m). This has several advantages:

- Since files are addressed via a content based key, all instances of a file are identifiable and accessible within to the system, regardless of naming and location.
- Parallel downloads allow load balancing over all nodes which carry an instance of a file.
- Partial downloads can be resumed at any time.
- The GFID can be used to verify the integrity of downloaded files.
- The separation of short-lived (HDB) and long-lived (NDB) meta-data allows for more efficient distributed implementations and caching strategies.
- If the GFID of the file is known to the user, no name-search is necessary to retrieve the file from the network. This e.g. allows for RAPP-URLs referring to a file to be embedded as links in HTML pages.
- Users aren't restricted to the generic RAPP search function, but can implement more efficient centralized or content specific name-databases.

- Since name-GFID mappings never get obsolete, NDB directories can also be distributed on permanent media.

2.2 Network Architecture

As RAPP is supposed to be a strict peer-to-peer protocol, no functionality must be delegated to centralized servers and all peers must – in principle – be able to perform the same tasks. This also excludes the use of external services like NTP or DNS or specialized servers to act as entry points to the system (i.e. new users must be able to connect to any peer).

The consequence from the above is, that peers have to maintain a network of connections to neighboring peers and have to ensure that the network graph remains coherent at all times. Other than coherence, RAPP shouldn't require a certain network topology to function. This esp. means that, while an average interconnection rate of 3 or above would be preferable, leafnodes and massive connected nodes should be allowed (e.g. to allow clients from a LAN to connect to the network through a peer running on a not-forwarding application firewall)

2.3 Design Principles

The overall design of RAPP should stick to the following principles:

1. **The KISS Principle:** Keep it small and simple (unless you have a *very* good reason not to).
2. **Flexibility:** RAPP has to consider and deal with many common annoyances of today's Internet, as firewalls, NAT, dynamic IPs, portfilters, etc. Victims of those measures should not be additionally discriminated against.
3. **Scalability:** There should be no inherent limits to the protocol neither in the number of users, nor in the number of files. Scalability has priority over completeness.
4. **Robustness:** Don't rely on anything, don't waste time if you have a fallback and never give up if you don't. Best effort is all we end up with anyhow, so don't bother to reach perfection unless it comes as a bargain.
5. **Laziness:** Remain passive and don't speak up unless you have to. This esp. means no keep-alive packets, no useless pings, no test packets, etc.

6. **Curiosity:** Evaluate every packet passed through you for useful information, including circumstantial data like delay times or packet loss.

7. **Caching:** Don't throw away information before you're sure you don't need it anymore.

8. **Fairness:** Do unto others as you would have others do unto you (and check on occasion whether they comply)

9. **Privacy:** RAPP isn't meant to provide real anonymity, but rather the anonymity of the mass. It is not a design goal to resist a determined attack with high level, long term traffic analysis, but it should provide you at least with plausible denyability in most situations.

2.4 Basic Communication Protocol

According to the principle "first make it work, then make it fast", RAPP should be based on a very simple basic protocol, which emphasizes robustness over efficency and works as a fallback for optional higher level protocol elements like an efficient search mechanism (see section 2.4.2), dynamic adaption of the network topology, social engineering, etc.

As RAPP is to be based on TCP/IP, this leaves us two choices for the transport protocol:

- Nodes with routable addresses open a TCP and a UDP port and use UDP to communicate among themselves.

- Nodes behind firewalls don't open any port and use TCP to initiate connections to their neighbors (which consequently have to be outside the firewall).

Just like gnutella, nodes can broadcast requests to their neighbors, which get propagated in a store and forward fashion throughout the network. A maximum-hops counter, which depends on the type of request and is decreased by at least one, each time a packet is forwarded, limits the maximum recursion depth and implements a "horizon". A time-to-live (TTL) field additionally limits the temporal scope of requests, helps to get obsolete packages out of the network and can be used as a means of flow-control. The TTL is also essential to indicate how long peers should remember forwarded packets as this allows the nodes to reclaim routing information from local data and also allows the system to sort out bogus reply packets which aren't related to a prior request.

However, it would be inefficient (and, since we use UDP, also risky with respect to packet loss) to forward replies all the way back through the incoming path, so each request also contains a field for a UDP return address. Peers who aren't able (or, for privacy reasons, aren't willing) to receive UDP replies themselves can leave the field empty, and any peer along the forward path can fill this field with his own UDP address, thereby effectively short-cutting the reply path. The very fact, that this possibility exists buys users plausible denyability: They can always claim the they're just receiving replies for another peer.

The above protocol can basically be used for all type of requests, like search queries, download requests or to look for peers who are willing to accept a new neighbor-connection. And while more efficient protocols, as e.g. for name-search or for directly requesting downloads from peers who are known to carry a certain file, can be additionally implemented, the above mechanism can always serve as a fallback, should the higher protocol fail.

2.4.1 Downloads

Download requests can also be broadcast in the above manner. All peers who have an instance of the requested GFID reply with an invitation packet indicating that they are willing to act as server for this file.

Servers with a public IPs include their IP and UDP/TCP port number in the reply, servers behind firewalls initiate a TCP connection to the port named in the request package (which must not necessarily be the originator of the request but, as explained above, can be any host along the request path who is willing to serve as a proxy for this download, which allows even peers behind two different firewalls to communicate, which otherwise would be impossible).

The client (i.e. the downloading peer) then requests unknown blocks of the file in a round-robin fashion from all servers while using the TTL field and the number of simultaneously requested blocks for flow control. The servers also serve block-requests in a round-robin fashion until the TTL of a request expires, in which case, any open block requests are discarded.

With any reply package, a server sends header information indicating how many blocks have been requested and how many have already been answered. The client can thereby estimate the packet loss on each connection and use this information for flow control and the scheduling of further block requests.

2.4.2 Distributed Name Search

One of the biggest challenges in designing RAPP is the implementation of an efficient search mechanism to find NDB records (i.e. filename-GFID pairs) for given name patterns.

While a broadcast-based mechanism as described in section 2.4 could be used for name search (like in gnutella), the scalability of this approach is limited by the number of queries each node can process.

As the latter is typically limited by the available bandwidth peers are willing to contribute (local memory and CPU time are orders of magnitude cheaper than bandwidth) and NDB-data is long lived, this suggests the implementation of caching/mirroring strategies. Besides the general design principles named in section 2.3, any solution would also have to meet the following criteria:

1. **Compatibility:** If a query can't be served, the basic communication protocol (s. section 2.4) should be used as a fallback.

2. **Controllability:** Any peer must (within certain bounds) be able to control how many local resources he is willing to contribute.

3. **Economy:** Only peers who intend to mirror data should take part in its distribution.

4. **Query Independence:** The implementation should not be restricted to certain types of queries.

A simple mechanism to meet the above criteria would be, to enable neighbors to mirror their filelists, so that they don't have to forward queries with a remaining hop-count of 1 and thereby reduce the overall network load of distributing a query by a factor equivalent to the average interconnection rate of the network graph. Recursive adaption of this method can further reduce network load at the cost of local workload, but would also require considerable startup costs, which can be an issue, as neighbor relations aren't permanent but prone to change as peers connect and disconnect to and from the system.

A more sophisticated method could be, to have every peer (or at least every peer with a public IP) mirror a well defined part of the global NDB; e.g. a peer could advertise that he is willing to mirror all names of GFIDs starting with a certain bitstring. Since a bitstring is relatively short, it could be made a part of the standard header for all RAPP packets which contain the host-address, so no explicit announcement mechanism would be necessary.

All peers monitor their connections for new mirrors; if one is found, peers which cache matching bitstrings synchronize their local databases, while all others check whether they have unannounced files with matching GFIDs, and if so, directly announce them to the mirroring host. Occasional reannouncements and resynchronizations as well as a limited TTL for database entries can ensure that unhosted files slowly disappear from the NDB and new entries get propagated throughout their appropriate mirrors.

All queries get tagged with an initially empty bitstring. When receiving a query, a peer first checks if he is mirroring the respective file-list and if so, answers the query, else he searches his local file-list for matches (ignoring the tag-string) and checks whether he knows about at least 2 mirrors who left-match the query tag, but differ on the following bit. If this is the case, the query gets forwarded to these hosts with the bit appended to the tag string; otherwise the query gets forwarded normally. To minimize the chance that this happens, new hosts can exchange an initial list of mirrors with the peers they connect to.

3 Goals

It is necessary to state that the above suggestions are just that – ideas which will have yet to stand the test of the realities which constitute the Internet as is is today, and will, once that the first prototype is out there, be under constant attack of the powers that be, e.g. ISP which try to limit their costumers' access, script kiddies' vandalism or hackers wanting to expose vulnerabilities of the system. The real challenge therfore is, to make the ideas presented above work in real life and to be flexible enough the adapt them, should this prove necessary.

My personal motivation to pursue a project like this is, besides the scientific value, the fact that – to the best of my knowledge – no such system is currently out there, and that I would really like to use such a program myself.

To make RAPP a reality, I suggest the following schedule:

1. Design of a minimal protocol which meets the principle design criteria presented above.
2. Prototype implementation of the protocol.
3. Publishing of the prototype, collecting feedback and experimental data (alpha testing).

4. Gradual improvement of the protocol and the prototype to address issues arising in the alpha test-phase.
5. Protocol feature freeze.
6. Development of a reference implementation.
7. Beta testing of the reference implementation.
8. Release of the final protocol spec.
9. Version 1.0 of the reference implementation.

Bernhard Ömer

Vienna, the 20^{th} *October 2000*