

# Communication in Intermittently-Connected Networks

by

Gregory Kempe

B.Sc., The University of the Witwatersrand, 2001  
B.Sc. (Honours), The University of the Witwatersrand, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

September 2005

© Gregory Kempe 2005

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) \_\_\_\_\_

Department of Computer Science

The University Of British Columbia  
Vancouver, Canada

Date \_\_\_\_\_

# Abstract

Communication in the face of intermittent, short-lived and possibly unreliable connectivity can be difficult when relying solely on Internet protocols, such as IP and TCP, which have an implicit assumption of well-connectedness. Furthermore, use of these protocols is impossible when there is no fully connected end-to-end path between hosts.

We present *Euonym*, an architecture that uses a layer of persistent names to identify hosts, networks and services, and allows arbitrary intermediate helper hosts to be interposed between endpoints on-the-fly. These helper hosts can provide routing, buffering and other support services to help relieve reliance on end-to-end paths. They can be placed to take advantage of intermittent connectivity when and as it arises and be used to supplement connectivity with untraditional networking and transport mechanisms, such as data mules and the postal network, without any additional support at the end hosts. We show that simple disconnection tolerance and use of intermediate hosts facilitates communication and promotes connectivity in intermittently-connected networks.

# Contents

<b>Abstract</b> . . . . .	ii
<b>Contents</b> . . . . .	iii
<b>List of Tables</b> . . . . .	vi
<b>List of Figures</b> . . . . .	vii
<b>Acknowledgements</b> . . . . .	viii
<b>Dedication</b> . . . . .	ix
<b>1 Introduction</b> . . . . .	1
1.1 Overview . . . . .	2
1.2 Outline . . . . .	3
<b>2 Intermittently-Connected Networks</b> . . . . .	4
2.1 Intermittently-Connected Networks . . . . .	5
2.2 Disconnection . . . . .	7
2.3 Example Scenario: Remote Community . . . . .	9
2.4 Making Use of Intermittent Connectivity . . . . .	11
2.4.1 Managing Disconnections . . . . .	11
2.4.2 End-to-End Paths and Intermediate Hosts . . . . .	13
2.4.3 The End-to-End Argument . . . . .	15
2.5 Summary . . . . .	16

---

<b>3</b>	<b>Euonym</b>	17
3.1	Architecture	17
3.1.1	Names	18
3.1.2	Name Stacks and Name Resolution	20
3.1.3	Establishing a Flow	21
3.1.4	Flow Identification	23
3.1.5	Source-Specified Routing	24
3.1.6	Late Binding and Re-binding	25
3.1.7	Intermediate Hosts	26
3.2	Implementation	27
3.2.1	API	27
3.2.2	TCP Interface	31
3.2.3	File Interface	34
3.2.4	Names: Format, Storage and Lookup	34
3.2.5	Limitations	36
3.3	Summary	36
<b>4</b>	<b>Using Euonym</b>	38
4.1	Batch Communication	38
4.2	Interactive Communication	40
4.3	Remote Service	44
4.4	Additional Discussion	46
4.4.1	Mobility	46
4.4.2	Switching to a Rendezvous Paradigm	47
4.5	Summary	47
<b>5</b>	<b>Related Work</b>	49
5.1	Disconnection Tolerance	49
5.2	Intermittently-Connected Networks	50
5.3	Identity and Addressing	52

---

<b>6 Conclusion</b> . . . . .	54
6.1 Future Work . . . . .	55
<b>Bibliography</b> . . . . .	56

# List of Tables

3.1	RESOLVE and RESOLVENAME algorithms . . . . .	21
3.2	Example client code . . . . .	29
3.3	Example server code . . . . .	30
4.1	Name mappings for the batch communication experiment . . . . .	40

# List of Figures

2.1	Example network: remote community . . . . .	9
3.1	Layering in Euonym . . . . .	18
3.2	Flows and connections in Euonym . . . . .	19
3.3	Basic example scenario . . . . .	22
3.4	Name stacks example . . . . .	25
3.5	TCP and the Euonym API . . . . .	32
4.1	Network layout and link availability for the batch communication experiment . . . . .	39
4.2	Data flow in the batch communication experiment . . . . .	41
4.3	Network layout and link availability for the interactive commu- nication experiment . . . . .	42
4.4	Data flow in the interactive communication experiment . . . . .	43
4.5	Data flow in the remote service experiment. . . . .	45

# Acknowledgements

I am indebted to my supervisor, Norm Hutchinson. His encouragement, support, insight and faith in my abilities are immensely appreciated and have taught me a great deal.

I am deeply thankful to my parents and brother, for their unquestioning love and support, and for coming half way around the world to visit me. The weekly games group provided many an entertaining evening and were willing victims of my culinary ventures. The Orphanage took me in and made me an honorary Orphan, a great privilege.

Thank you to Dustin, Micheline and Sarah for making such an indelible mark upon my life and showing me how to appreciate the simple things.

GREG KEMPE

*The University of British Columbia*

*September 2005*

For the Three Blind Mice, and Tina, Ben and Leo.

Tea, though ridiculed by those who are naturally  
coarse in their nervous sensibilities . . . will always  
be the favourite beverage of the intellectual.

*Thomas De Quincey (1785–1859)*

Send three and fourpence, we're going to a dance.

*Unknown*

# Chapter 1

## Introduction

We usually expect a connection in the Internet to exist at the time of, and for the duration of, its use; we do not attempt to visit a website half an hour before we connect to the Internet, or expect a file transfer to resume after more than a few minutes of our wireless connection being lost. Synchronous communication in the Internet requires that both hosts be online at the same time. Some networks, however, lack such luxuries and these expectations become unreasonable.

Networks in remote or inaccessible areas and parts of the developing world lack the infrastructure that is taken for granted in the well-connected Internet. As a result, they are only occasionally connected to other networks, or the connection may be irregular or unreliable. In these environments, it is important to reduce the impact of disruptions and make as much use of connections, when they arise, as possible. This is especially important if a device has limited power, transmission range or lifetime. It may also be necessary to augment Internet-style connectivity with more unconventional data transmission mechanisms, especially when they are the only feasible connection options.

This thesis presents *Euonym*,<sup>1</sup> an application-agnostic architecture that addresses these issues by allowing intermediate hosts to be explicitly interposed on the path between two endpoints. The architecture helps support communication in heterogeneous, intermittently-connected networks that lack connected end-to-end paths and preserves connections across link failures and network address changes.

---

<sup>1</sup>*euonym*: a name well-suited to the person, place or thing named.

---

## 1.1 Overview

While we often take ubiquitous, high-quality Internet connections for granted, many parts of the world are not so well connected. Schools in outlying regions of South Africa, for example, often cannot afford the long-distance phone calls required for dialup Internet [30], and nomadic reindeer herders in northern Scandinavia must rely on intermittent long-distance radio and traditional mail to provide contact with other herders and larger towns [23]. Network connections can be limited, short-lived and far-between. To compound the problem, transport protocols like TCP and UDP fail in the face of excessive delay or, more generally, the lack of an immediately available end-to-end path. This makes them unsuitable for use in networks where such a path doesn't exist or doesn't exist for the duration of a connection.

Given the possible rarity of connection availability, we would like applications to be able to make the most of connection opportunities whenever they arise, without necessarily having to know about them in advance. We would like to remove the dependence on end-to-end paths and allow new links and network types to be added to an existing connection, even if they are unsupported by the end hosts.

The Euonym architecture uses intermediate hosts to help achieve these goals. Intermediate hosts can perform a number of support functions—such as buffering, routing and acting as rendezvous points—and help to bridge heterogeneous network elements and allow novel transport mechanisms to be used in conjunction with traditional ones.

In this thesis we describe the problems associated with intermittently connected networks and outline ways that intermediate hosts can help mitigate them. The architecture we present uses a layer of names above IP addresses to persistently identify hosts and allow intermediate hosts to be explicitly involved in a connection. We demonstrate that it improves use of existing, intermittent connectivity and allows connections to be established where none would have been possible before.

The contributions of the work are twofold, namely that

1. intermediate hosts can be explicitly included in a path between two endpoints to facilitate communication in intermittently-connected networks; and
2. the separation of network address and host identity allows end hosts to delegate responsibility and operations to intermediary hosts that are better equipped to make use of, or improve upon, connectivity.

## 1.2 Outline

The rest of the thesis takes the following form. In Chapter 2 we discuss intermittently connected networks and the challenges they pose, and describe using intermediate hosts to support communication. The Euonym architecture and implementation are described in Chapter 3, and example uses of it are detailed in Chapter 4. Chapter 5 discusses related work and in Chapter 6 we briefly outline future directions for our work and conclude.

## Chapter 2

# Intermittently-Connected Networks

The idea for our work has its origins in the nascent field of the interplanetary Internet<sup>1</sup>. The IRTF's now-historical Interplanetary Internet Research Group was created to investigate the technical aspects of extending the terrestrial Internet into space. The expansive distances and delays involved in interplanetary and deep-space communication present significant challenges to this movement, one of them being that widespread Internet protocols like TCP perform extremely poorly (if at all) in the face of such delays. The group's work formed the foundation for the Delay Tolerant Networking Research Group [3], which is investigating an architecture for bridging and traversing heterogeneous networks, including those with long delays and intermittent connectivity. The solutions and approaches formulated by these groups have many applications back here on Earth where, even on much more human scales, there are challenges to connecting disparate and far-flung networks.

In this chapter we describe some of these challenges and ways of tackling them. We begin with a discussion of the networks and the characteristics that make them difficult to work with. Next, we describe approaches to making them usable, focusing on intermediate helper hosts and ways of involving them in a connection.

---

<sup>1</sup>See the Interplanetary Internet Special Interest Group at [www.ipnsig.org](http://www.ipnsig.org).

## 2.1 Intermittently-Connected Networks

The growth of the Internet and our reliance on the services it provides has prompted its expansion into areas and situations that do not support its architectural models. From deep space networks [12] and isolated nomadic community networks [23], to school LANs in remote areas [30] and use of the postal network as part of the Internet [29], all these networks break some of the Internet's (implicit) performance and architectural assumptions. Specifically, the communication model of the Internet relies upon

- low network latency, on the order of milliseconds;
- continuous connectivity over the period of communication;
- bidirectional communication; and
- connected end-to-end paths.

These requirements are often unstated but protocols like TCP and UDP, as well as those that rely on them, fail when they are not met. Fall [5, 6] calls the class of networks and inter-networks that do not support these assumptions *challenged networks*.

To illustrate, consider the following scenario. The Sámi Network Connectivity Project [23] is working on providing Internet connectivity to reindeer herders in the Sámi region of northern Scandinavia. The herders are nomadic, following their reindeer's yearly migration cycle, and the region in which they move is remote and isolated and lacks reliable wired, wireless or satellite communication. Members of the community have access to their own local network but getting beyond it is difficult. The community-wide network and its connection to the Internet are an example of a challenged network.

Opportunistic use of periodic connectivity would greatly increase their interaction with other herders and the outside world. Potential contact opportunities include brief satellite and long-range radio connection windows, people and vehicles that journey between the communities and larger towns that have better

infrastructure, and postal services that transport traditional mail. A number of these could also be used in combination to connect extremely remote herders with their home community and then the Internet.

There is more than just a physical difference between these connection options. They cover a significant range in bandwidth and delay and each may require a different transport protocol as well as physical and link-layer protocols. Different link technologies must be sewn together and routes chosen based on current connections, as well as upcoming opportunities, and their associated economic costs.

This example highlights three causes of difficulty when working with challenged networks which conflict with the assumptions stated above:

- connection interruptions,
- lack of infrastructure, and
- heterogeneity.

The first two result in intermittent connectivity and the third makes interoperation difficult. They may exacerbate each other, too; a fragile heterogeneous infrastructure can result in service interruptions. Consider these from the perspective of a TCP-based application that must traverse a link in a challenged network to contact a remote host. At the transport level, all three result in delivery failures which finally lead to a TCP timeout and disconnection, producing an error at the application level.

In general, we characterise *disconnections* as breaks in the transmission of data of sufficient duration that they cannot be overcome by the transport protocol. The result is usually transport- and application-level errors. In addition, TCP requires a *connected end-to-end path* between the hosts for the duration of the connection. This is a path between two endpoints that experiences no disconnections across its entire length.

These two concepts are our focus points for supporting communication in intermittently-connected networks. We wish to reduce the impact of disconnec-

tions on applications and establish connected end-to-end paths or remove the reliance on them. In doing so, we aim to overcome the three causes of difficulty by

- tolerating and working around connection interruptions,
- providing a general means of using new, possibly unconventional infrastructure to create or augment a network, and
- helping heterogeneous elements in the network to cooperate.

The rest of this chapter discusses approaches to achieving these goals by focusing on disconnections and end-to-end paths, and how they relate to each other.

## 2.2 Disconnection

A disconnection is usually seen as a binary absolute: either there is a connection to a remote host and it can be communicated with, or, to all intents and purposes, the host does not exist. Instead, we view lack of connectivity as a continuum and divide it into three broad types. We discuss them in order of increasing complexity, as characterised by the responses required of the user, the application and the network itself.

**Intermittent disconnection** Disconnections are infrequent and short-lived, on the order of minutes or a few hours. For example, a host losing contact with one wireless access point before entering the range of another, or a laptop entering a low power mode over night. Overcoming these disconnections is comparatively straightforward: increase the transport protocol's timeout period or introduce a method for resuming connections. They are transient and relatively brief and there is no need for either the user or the application to be involved when they occur; they should be handled entirely at the network level.

Managing this form of disconnection is an area of active research and is not a focus of this thesis. We briefly describe the work, insofar as it relates to ours,

in Chapter 5. The thesis focuses instead on the next two categories which are only beginning to be investigated.

**Long-lived disconnection** This category involves networks that are predominantly disconnected. Connections are brief, few, and far between, and hosts must make as much use of them as possible. When available, though, they are readily usable: the delay and bandwidth are such that regular Internet protocols can be used effectively. An example of this category is a remote community that has an Internet connection for only a few hours every week.

Applications must be aware of these disconnections and should provide users with feedback on long-lived operations. They must be ready to use connectivity when it arises and be able to work around long delays and periods of disconnection. Lower-level protocols should support them in this by tolerating disconnections and delays and allowing old connections to be resumed, possibly even across application and system restarts.

**Systemic disconnection** The path between two hosts has many disconnected components and periodic links, and there is no guarantee of a connected end-to-end path. For example, it may be hindered by a component that experiences extreme delay or is even missing completely.<sup>2</sup> Both end hosts and hosts along the way must overcome their own separate disconnections—possibly in entirely different ways—and each has its own period of visibility. In contrast with the second category, even when a connected path exists it may not support Internet protocols. If the two hosts do not have overlapping connection windows, or a link has excessive delay, Internet-style synchronous communication is impossible.

Applications, users and the network must all respond to this form of disconnection. In addition to the requirements for long-lived disconnections, applications should favour batch operations and protocols over chatty, interactive, round-trip dependent ones. The user may need to make complex routing deci-

---

<sup>2</sup>Lack of a network link is, after all, simply a long-lived disconnection waiting for a suitable physical transport to bridge it.

sions that use local as well as global information such as upcoming connection windows. Applications should allow users to provide not just the names of remote hosts but also rendezvous points or routing information in order to make use of these possibly distant connections. The network must support disconnections as per the previous two categories, as well as allow adjacent, heterogeneous networks to interoperate.

### 2.3 Example Scenario: Remote Community

We exemplify these three categories with a detailed example scenario, that of a student working in a remote community. The example is inspired by the Wizzy Digital Courier project [30]. The project provides intermittent connectivity to isolated schools in South Africa by buffering web and email requests until off-peak hours when a dialup Internet connection is cost-effective.

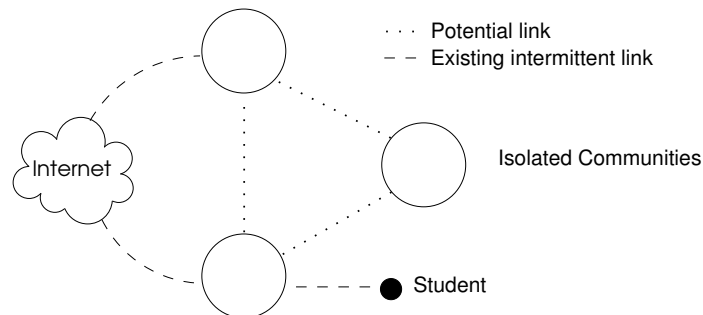


Figure 2.1: Student in a remote community with intermittent connections to the Internet. There are potential inter-community links that may rely on untraditional transport methods.

Consider a student doing research in an isolated community, Figure 2.1. The student's laptop is connected to the community's small LAN, centred at the local school. Long-distance phone call costs are such that a dialup connection to the ISP in the closest town is practical only over the weekends; the network remains isolated during the week. The student periodically makes trips to nearby

communities where other students are performing related work and have similar networking setups. He is disconnected while in between communities and, even when in them, he is largely isolated from the Internet. On longer, data-gathering field trips into the countryside he is disconnected from all networks.

Whenever a connection is available, the student streams his data back to a server at his university where computationally expensive processing is performed and the results sent back to him. He also remotely runs interactive applications on the local LAN server and on the university server and needs to keep in contact with other students in the nearby communities.

The student's task is an unenviable one when relying on applications dependent on Internet protocols and end-to-end paths, such as ftp, scp and ssh. Every weekend when the Internet connection is available he must restart his applications and re-establish his outside connections, each in an application-specific manner. Incomplete ftp transfers must be resumed, aborted scp transfers restarted from scratch, ssh connections re-established and applications relying on ssh-tunneled data restarted. Late every Sunday night his connections are ungracefully terminated, remote interactive applications are disconnected and local applications produce an error when their ssh tunnels are destroyed. The same problems occur with his local connections when he leaves the community LAN. Despite being physically near to each other, the two communities are isolated during the week and remaining in contact with the other students is equally troublesome. They are forced to rely on a third party network (the Internet) in order to communicate.

This scenario has elements of both long-lived disconnections and systemic disconnections. The community experiences periods of long-lived disconnection but, when their Internet connection is available, it supports Internet protocols and synchronous communication. If the student is within the community during these times, he is equally well connected and can operate as just another host in the Internet. If he is away, however, the disconnections become systemic. He is now disconnected from both the community and the rest of the Internet. The periods during which the community is connected may not coincide with

periods when he is connected to the LAN. Direct inter-community connections are not possible at all.

As with the Sámi community network, there are a number of potential, unexploited connection opportunities. There are regular commuters and buses travelling between the two communities and the town. There is a regular postal service in the region and, although expensive, a periodic high-bandwidth satellite link is also available. We would like the student (and the rest of the community) to be able to take advantage of these options, despite differences in bandwidth, delay and availability.

## 2.4 Making Use of Intermittent Connectivity

We now discuss two aspects of supporting connectivity in challenged networks: tolerating disconnections, and using intermediate hosts to support communication when disconnections prevent connected end-to-end paths.

### 2.4.1 Managing Disconnections

Making applications and networks disconnection-tolerant is an aspect of handling intermittent disconnections which, as we stated previously, is already a well-researched area. We outline the general concepts here and briefly discuss related work in Chapter 5.

Tolerating disruptions mid-way through a conversation requires that the communication abstraction meets two requirements: a break mid-way must not be fatal, and it must be possible to resume the conversation when the disruption passes. In other words, treat the disconnection as a transient failure, rather than a permanent one.

Some applications support connection resumption at a logical or operational level. FTP [21], for instance, can resume a file transfer part way. SMTP [15] is specifically designed around a disconnection-tolerant store-and-forward architecture. These approaches are application-specific, place the burden on the

application and generally treat the disruption as an exception. There are trade-offs involved when choosing where to manage disconnections, however. The closer management is moved to the network the less control and input the application has over the management. Moving it closer to the application can complicate the application unnecessarily and duplicate code and functionality.

These tradeoffs are related to the three types of disconnections discussed in Section 2.2 and emphasise the differences between them. Short, intermittent disconnections that can be readily managed at the network level without the application's involvement should be, especially if the application does not need to adjust its behaviour in response to the disconnection. In contrast, long-lived and systemic disconnections should not be abstracted out entirely. The application and user often know more in these cases than it is reasonable for the network to know, and should be involved in the handling of the disconnection. A session layer can be used to fill the middle ground between these options, providing a range of functionality to the application with support from the transport and network levels. Complex disconnections often go beyond the abilities of the network and transport level to handle independently in any case. Involving the application also makes it easier to resume conversations across application and system restarts since it has more control over what session state is saved.

When connection information is associated with a host's network address, as is the case with TCP and UDP, managing disconnections is complicated further when an address change is involved. Associating the information with a persistent host identity instead helps make low-level address changes transparent.

The issue of separating host identity from address has been raised periodically over the last few decades, but only recently has it become a focus of the research community as the demand for mobile communication grows; work on services like Mobile IP [20] and Mobile IPv6 [14] aim to mitigate the effects of these changes in TCP/IP networks. These issues must be taken into account in intermittently-connected networks where address changes can be both causes and side effects of network disruptions.

Gracefully handling disconnections is only the first step towards communicating using intermittent connectivity. We must still reconsider the reliance on end-to-end paths.

### 2.4.2 End-to-End Paths and Intermediate Hosts

Hosts require a connected end-to-end path only when they must take all responsibility for the data they send. In a well-connected network, this is a reasonable expectation and is the basis for the Internet's end-to-end argument [24]. Connected end-to-end paths and disconnections are two conflicting concepts, however. Reliance on end-to-end paths becomes problematic when data reaches a disconnection in the network and the end host does not have sufficient information (or control) to work around the disruption. Moreover, those hosts closer to the disconnection that are better informed and able to handle it are not in a position to do so. If responsibility for the data can be passed to these intermediate hosts, the reliance on end-to-end paths can be relaxed.

Consider the heterogeneity of the infrastructure in intermittently-connected networks. Trying to formulate a one-size-fits-all transport protocol is an exercise in futility; different links and networks have different characteristics which should determine the behaviour and design of their protocols. It is difficult for an end host to switch to a different transport or network protocol if it is distanced from the link that needs it. It is even more difficult if the data must traverse a series of heterogeneous links. By allowing each network and link to choose its own protocol and using intermediate hosts to sew the links together at their edges, we can tackle a diverse range of link types and technologies without relying on a single, all-encompassing transport.

Intermediate hosts can do more than act as translators and remove the reliance on end-to-end paths. Explicitly including them in a connection makes them a very powerful and flexible means of providing services useful in challenged networks, especially those that are location-dependent. We can use them to provide functionality on a per-need basis, rather than attempting to engineer

a one-size-fits-all service model for such greatly varied networks. This functionality includes:

**Routing** An intermediate host can make high-level routing decisions when there are multiple connection opportunities. A network may have a relatively cheap but slow dialup Internet connection; a fast, expensive, high-bandwidth satellite link; and a free, high-bandwidth, high-delay data courier all available at different times. Routing decisions can be complex and must take into account current as well as up-coming connections and their throughput and cost, and it is likely that they will be made by a human operator rather than automatically. Where possible, unexpected connections should be used opportunistically as they arise. Making these decisions at a single point is much simpler than at multiple end hosts.

**Forwarding points** Intermediate hosts can act on behalf of end hosts when the latter are unavailable. In our example network, while the student is out in the field and disconnected, he can use an intermediate host in the community LAN as temporary storage point for incoming data. Upon his return, the intermediate host can forward the buffered data to him. A host can be similarly interposed as a forwarding point for outgoing data, performing buffering in the opposite direction. The student can send data in expectation of an up-coming connection and then disconnect from the network. When a connection between the intermediate host and the next downstream host (possibly another forwarding point) is established, the data can be sent without the student being involved.

**Rendezvous points** In situations where an intermediate host cannot actively connect to an endpoint (or another intermediate host) due to network configuration, firewalls etc., it can act as a rendezvous point instead of a forwarding point. The intermediate host buffers data in either direction and passively waits for an end host to contact it and establish a connection along which to forward

data. The result is a *put/get* communication model that naturally does not rely on a connected end-to-end path.

**Message consolidation** It can be useful to group multiple streams into a single stream that is then transported *en masse*. This is important if a link is only cost effective above a certain utilisation level. Rather than fitfully sending data as it becomes available, a host can buffer data from multiple sources until a critical mass is reached and then pass it on to the intermediate host responsible for the link. On the far side, the streams are separated and continue their separate journeys.

In general, intermediate hosts allow responsibility for data to be delegated to other hosts in the network. This allows the end host to take advantage of connections and services that it would not normally have access to, and naturally supports cooperation of heterogeneous network elements.

### 2.4.3 The End-to-End Argument

Intermediate hosts are certainly not new in the Internet. Middleboxes such as NAT devices, web caches and proxies are also aimed at providing services, such as performance improvements and address translation, to end hosts at specific points in the network. They have been widely criticised for violating the end-to-end argument and fate sharing principle to the detriment of the network [2]. They complicate network management and protocols and can have unforeseen effects on network services. These problems are mainly due to their attempts to be transparent. They intercept and modify data which is not addressed to them and operate without the end host's knowledge or cooperation. For instance, NAT devices adjust the IP addresses of packets they intercept, interfering with the identity semantics associated with addresses (issues of identity and address are discussed further in the next chapter).

It is desirable that we involve intermediate hosts without falling into the same trap. Work such as [8, 28, 7] has explored the possibility of making middleboxes

---

first-class objects in the network in an attempt to reinstate the argument. We take a similar approach and argue that because the end hosts explicitly invoke the intermediate hosts, they do not violate the argument. Both hosts are aware of each other and end hosts have control over which intermediate hosts are involved in a connection and when. Furthermore, our intermediate hosts are not trying to “trick” the end hosts or modify transport- or application-level data. As we discuss in Chapter 3, our architecture explicitly terminates transport-level connections at intermediate hosts and application data must arrive unmodified at its destination. It is permissible for intermediate hosts to establish state because they are so involved in a connection. As a result, they share the fate of the connection and the end hosts.

## 2.5 Summary

The class of *challenged networks* are those that do not support all of the architectural assumptions of the internet. They may lack end-to-end paths, experience frequent disruptions, have excessive delay and bandwidth limitations, and involve heterogeneous network elements. To facilitate communication in these networks and simplify application development two issues must be addressed: disconnections and lack of a connected end-to-end path between hosts.

Intermediate hosts can help address these issues. By explicitly including them on a path, they remain in accord with the end-to-end argument and share the fate of the endpoints and the application-level flow. They are useful tools for providing functionality to the network and the end hosts without an over-engineered architecture that provides an exhaustive suite of services.

In the next chapter we describe the Euonym architecture and its implementation. The architecture supports connection resumption and explicit involvement of arbitrary intermediate hosts on the path between endpoints.

## Chapter 3

# Euonym

The Euonym architecture uses intermediate hosts to provide services to end hosts in challenged networks. We use a layer of persistent names to identify hosts and services in the network and provide a basic form of disconnection tolerance. End hosts can use source and destination-specified routing to include intermediate hosts in a connection, invoking their services only when needed. The architecture supports arbitrary network address formats without unnecessary complexity at the end hosts and, by using late binding, delays the interpretation of names and addresses until they can be done at a suitable point in the network. The architecture describes how intermediate hosts are named and should participate in a connection, but does not impose a specific set of services or assumptions on them or the end applications.

In this chapter, we describe the Euonym architecture in general and the specifics of our example implementation.

### 3.1 Architecture

Euonym lies between the application and the transport layer (or session layer, if present)—Figure 3.1. We term a logical connection between the source and final destination hosts a *flow*, and the transport-level connections between these hosts and intermediate hosts *connections*. Connections are always explicitly terminated at intermediate hosts.<sup>1</sup> A flow is always one-way while the underlying connections may be two-way. To establish bi-directional communication, the

---

<sup>1</sup>In other words, we are not proposing a transport protocol that involves intermediate hosts.

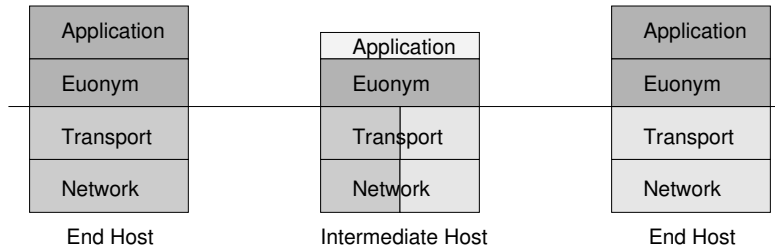


Figure 3.1: Layering in the Euonym architecture. Transport and network connections terminate at an intermediate host, while Euonym flows and application-level connections span them. An intermediate host generally has a helper application which is not associated with the end host applications.

end host applications must each initiate their own flow to the other party. The architecture is depicted in Figure 3.2.

### 3.1.1 Names

Every Euonym host has one or more *names*: flat (non-aggregable), opaque, globally unique identifiers, based on the endpoint identifiers described by Balakrishnan *et al.* [1]. When necessary, a host's name is translated into an *address* for use by the transport and network layers, such as an IP address.

As we describe in the next few sections, a Euonym name can identify more than just a single host. It can identify a path that data must take, a service provided by an intermediate host, a group of hosts, or even an entire network. The meaning of name is dependent on where it is resolved and how it is used. For instance, an application can identify a host in a remote network simply using the host's name. Initially, the name corresponds to a gateway to use or a route to follow to reach the remote network. Once in that network, it corresponds to the services needed in order to reach the end host and, finally, the local address of the end host, whatever form that may take.

This flexibility of interpretation is an important aspect of the architecture. We do not wish to encumber a name with semantics such as location, admin-

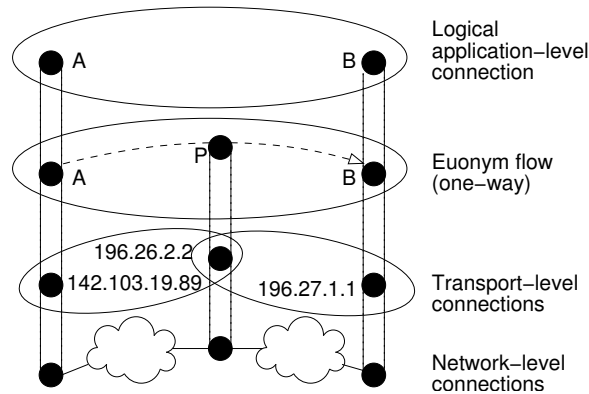


Figure 3.2: Flows and connections in the Euonym architecture, with end hosts A and B and intermediate host P. Euonym *flows* are one way and transport-level *connections* are usually two-way.

istrative domain, network membership or host type. For instance, given that our challenged networks are often divided into isolated regions, it is tempting to identify a host using its region name along with its network address. This simplifies routing but complicates management, especially since regions are likely to be ephemeral and hosts may move frequently between them. Any aspect of a host that changes during its lifetime without affecting its identity should not require a change of name. The dynamic nature of challenged networks also means that we must avoid any form of hierarchy, be it implicit or explicit: hosts are likely to move within any hierarchy as the network changes, and name lookup and storage must be done in a distributed manner because centralised services (such as DNS) do not lend themselves to these networks.

The alternative to opaque names requires that we identify namespaces by separating a host identifier into a {type, identifier} pair and use the type portion to determine the identifier's interpretation. This has two major drawbacks: it imposes the overhead of managing a type namespace, and it prevents context-dependent interpretation of a name. Instead, we use a single common namespace and let the context and use of a name determine its interpretation.

## Security

Separating a host's identity from its network address introduces security issues that are beyond the scope of this thesis. Briefly, naming a host with its address implicitly provides a degree of security because data is routed using the name, and a third party must be on the route in order to intercept it. This security is lost when data is routed based on address alone and identity must be verified separately. Nikander *et al.* [19] discuss these and related security concerns in the context of the Host Identity Protocol [18]. The HIP architecture uses similarly flat hostnames and, by embedding the cryptographic hash of a host's public key in its name, they make the name self-certifying. This allows the identity of a remote host to be verified relatively easily. A similar approach could be taken to secure the Euonym architecture.

### 3.1.2 Name Stacks and Name Resolution

In addition to an address, a name can also map to a *name stack*: an ordered list of names and addresses. Name stacks are an essential part of the Euonym architecture. They give it the power of source- and destination-specified routing, late binding, and are used to involve intermediate hosts in a flow.

Before connecting to a remote host, an application must resolve the remote name into an address suitable for the transport layer. It does this by resolving a name into a name stack using the `RESOLVENAME` function (Table 3.1). The `LOOKUP` operation simply finds the mapping between a name and its value: either an address, another name, a (possibly empty) name stack, or *null* if no mapping exists. The lookup is performed by a naming service specific to the local network, such as a DHT, in which hosts have published their name mappings.

An empty result stack and a stack with *null* at the top indicate errors. Note that when the mapping of a name is pushed onto the stack, it is pushed above the name itself. Thus, a record of the path taken to resolve a name to its eventual address (or error) is preserved within the result. Additionally, a name

---

```

function RESOLVENAME(name n)
    push n onto a new stack s
    return RESOLVE(s)
end function

function RESOLVE(stack s)
    set c to peek(s)
    if c is an address or null then return s
    set m to LOOKUP(c)
    if m is empty then pop(s)           ▷ An empty stack pops the name
    else push m onto s end if
    return RESOLVE(s)
end function

```

Table 3.1: The RESOLVE and RESOLVENAME algorithms for name resolution.

that maps to an empty stack is treated differently: the name is popped and the resulting stack is resolved. This allows a name to be removed from the stack when it is no longer needed. For instance, a network’s name resolves to an empty stack within the network itself as nothing needs to be done to reach it.

Names and addresses can be mixed in a name stack and we must be able to distinguish between them without imposing limitations on the format of network-dependent addresses. We could explicitly separate them into Euonym and non-Euonym namespaces but, given that an address should only be used in networks and by hosts that can understand it, this seems overly complex. Instead, we separate them implicitly and simply require that a host that uses a certain type of address can distinguish it from a Euonym name.

### 3.1.3 Establishing a Flow

We now describe how a Euonym host initiates a flow to another host and invokes the services of intermediaries. We start with a basic scenario involving the stu-

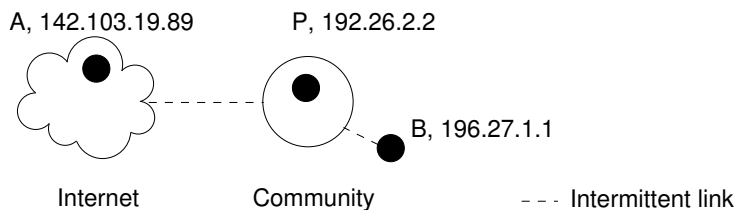


Figure 3.3: Example scenario:  $A$  is the university server,  $B$  is the student’s laptop and  $P$  is the intermediate host that buffers data destined for  $B$ .

dent working in a remote community, as described in Chapter 2. For simplicity, we assume that all endpoints use IP addresses and TCP as a transport, but the general case follows easily. The configuration is shown in Figure 3.3.

The student wishes to interpose host  $P^2$  on any flow destined for his laptop,  $B$ . He does this by making  $B$ ’s name map to the stack  $\{P, 196.27.1.1\}^3$ . The names  $A$  and  $P$  each map to their host’s IP address. When  $A$  initiates a flow to  $B$ , it recursively resolves  $B$  using `RESOLVE_NAME` to produce the stack  $\{196.26.2.2, P, 196.27.1.1, B\}$ .  $A$  pops the top entry (196.26.2.2) and opens a TCP connection to it. As part of the connection handshake,  $A$  sends the name of the flow’s origin (itself) and the modified stack to  $P$ . When  $P$  receives the stack, it pops off the first entry and verifies that it matches its own name. Since there are additional entries on the stack it knows that it is an intermediate host and must forward the data onward. It may also check the remainder of the stack and ensure that it is willing to forward data toward the final host.

$P$  then performs a similar operation to complete the flow, with only a slight modification. Since it already has a stack of names to work with—namely  $\{196.27.1.1, B\}$ —it can bootstrap the resolution process. It calls `RESOLVE` with this stack as an argument and, since the top is an IP address, the call returns without modifying it.  $P$  pops the top entry and connects to it, sending the origin host ( $A$ ) and the modified stack (now just  $\{B\}$ ) in the handshake. As before,  $B$  pops the top off the stack and confirms that it is the appropriate

<sup>2</sup>For readability, we use these names instead of Euonym’s 64-bit identifiers.

<sup>3</sup>Stacks are written with the top at the left and the bottom on the right.

target for the connection. Furthermore, since the stack is now empty,  $B$  knows that it is the final endpoint for the flow.

If  $B$  needs to send data to  $A$ , it establishes a new flow with  $A$  as the destination using the process described above. Recall that while each individual connection may be two-way, a flow is one-way. Depending on how  $A$  maps its name, the return path from  $B$  to  $A$  may differ from that followed from  $A$  to  $B$ . Different routes, services and intermediate hosts may be needed on the reverse path. This may be the case if flow components are naturally asymmetrical or one-way (e.g., a link served by a data courier on a circular route) or if the return flow is established some time after the initial flow, when the network topology and connectivity situation has changed. Both parts of a two-way connection can be utilised if the hosts are in a position to do so, but in general we allow the asymmetry of flows so as not to limit the use of the architecture.

### 3.1.4 Flow Identification

At both intermediate and end hosts, a flow is identified by its origin (a name) and its destination (a name stack), as received from the upstream host or specified by the application. In the above example, the flow is identified at  $A$  by  $\langle A, \{B\} \rangle$ , at  $P$  by  $\langle A, \{P, 196.27.1.1, B\} \rangle$  and at  $B$  by  $\langle A, \{B\} \rangle$ .

Note that there is no impact on downstream hosts, including the destination, if a non-originating upstream host changes. That is, if a connection goes from  $A$  to  $B$  via  $P$ , then the same connection state at  $B$  is used if  $P$  is removed or replaced, or another host is interposed on either side of  $P$ . Since names are removed from the stack as it progresses along the components of the flow, downstream hosts are ignorant of any components (and hence any changes) upstream of them. Thus, a host can easily receive data for the same flow from multiple upstream hosts.

Consider the impact of this naming scheme on a flow's final endpoint (at the risk of being repetitive, let us call it  $B$ ). If hosts are added, replaced or removed upstream, there should be no change at  $B$ ; it still sees a flow from  $A$  to  $B$ . If

---

hosts are added after  $B$ , however, it becomes an intermediate host for that flow and not a destination, even if the origin remains the same. New connections to  $B$  that have this new name stack are multiplexed separately from the original flow's connections. The result is that  $B$  now sees two separate flows.

End hosts can be seen as a degenerate form of intermediate host. Each intermediate host has an incoming half and an outgoing half. The incoming half waits for connections from upstream hosts and passes the flow origin and name stack to the outgoing half, which connects to downstream hosts. A flow's origin host is simply an intermediate host with only an outgoing half that receives the destination name from the application. Similarly, the final end host is an intermediary with only an incoming half. This suggests that an origin host could specify a name stack as a destination, instead of just a name. This gives us source-specified routing, which we discuss next.

### 3.1.5 Source-Specified Routing

We have seen how a destination host can invoke an intermediate host on incoming flows—destination-specified routing, which applies equally well to intermediate hosts.

An origin host can use source-specified routing to invoke services on outgoing flows. Instead of specifying just a destination name, it specifies a name stack that identifies the services to use. For example, to include service  $Q$  on a flow from  $A$  to  $B$ , the application at  $A$  specifies  $\{Q, B\}$  as the flow destination. Name resolution and flow establishment proceed as before, with the exception that  $B$  will be resolved by  $Q$  instead of  $A$ . This can also be performed on a per-host, rather than a per-application basis, by storing custom bindings locally on a host. In this case, to interpose  $Q$  on all flows destined for  $B$ , host  $A$  locally maps  $B$  to  $\{Q\}$ .

Again, intermediate hosts are no exception. They can push names onto the stack used by their outgoing half before the resolution stage. This does not interfere with flow multiplexing, which is performed by the incoming half before

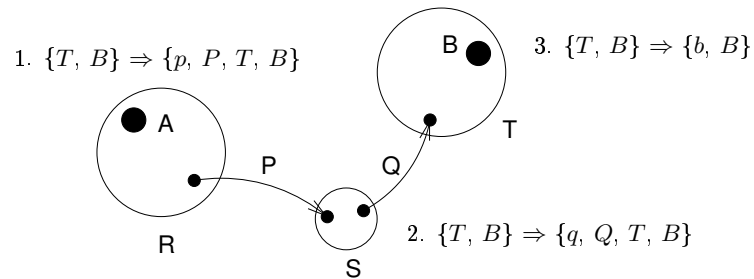


Figure 3.4: Source-specified routing, late binding and re-binding, and name interpretation.  $A$  in network  $R$  contacts  $B$  in network  $T$  via network  $S$  and intermediate services  $P$  and  $Q$ . Host  $X$  has address  $x$ .

the stack is changed. In addition, flow identification at those downstream hosts already on the stack is unaffected.

### 3.1.6 Late Binding and Re-binding

A host might wish to specify two successive intermediate hosts but delay the binding of the second one's name until after the first has been invoked. Often, this is because a name is location dependent. If the first name is a service that links two isolated networks, resolving the second name in the flow origin's local network may produce an empty result. Instead, resolution of the name must be delayed until it can be done inside the second network.

Alternatively, a single name with different mappings can be used. In the first network, the name resolves to the path required to reach the second network. Once there, it resolves to the end host itself.

The next example (Figure 3.4) demonstrates source-specified routing, late binding and re-binding of names, and interpretation of names as hosts, services, paths and networks, depending on the location.

Host  $A$  in network  $R$  wants to contact host  $B$  in network  $T$ . To do so, it must go via network  $S$  using the services  $P$  and  $Q$ . The flow destination is specified by the application as  $\{T, B\}$ . The name  $T$  is interpreted not as a network name but as a path to follow to reach the network: service  $P$  which

joins  $R$  and  $S$ . The destination is resolved by  $A$  to produce  $\{p, P, T, B\}$  (where  $p$  is the network address of  $P$ ). Once in network  $S$ , the name  $T$  is re-interpreted as a method to reach  $T$  using service  $Q$ . So  $\{T, B\}$  is resolved to  $\{q, Q, T, B\}$  and network  $T$  is reached. Within  $T$  itself, the name  $T$  maps to an empty stack and so is popped. Finally, the resulting stack  $\{B\}$  is resolved to  $\{b, B\}$  and the flow is completed.

### 3.1.7 Intermediate Hosts

#### Failures and Fate Sharing

Interposing hosts into a flow introduces a degree of complexity and poses some challenging issues. As discussed in Chapter 2, we feel that since these hosts are explicitly involved in the flow they do not violate the end-to-end argument. Even though their inclusion is transparent to the application, the end hosts are still aware of them. They are not hosts that perform operations on arbitrary connections without control, but rather are knowingly invoked by a host that *is* involved in the flow.

The issue of fate sharing and just how involved the intermediate hosts are is somewhat less clear. If an intermediate host fails, should the entire flow fail? In some cases such a failure may result in unrecoverable data loss which prevents the connection from continuing or being recovered. In other cases, it is possible that the end hosts can resend lost data and recover gracefully. Either way, it seems prudent to relegate this decision to upstream hosts and, eventually, the origin host itself. That is, if an intermediate host detects an unrecoverable failure on a connection, it should cascade the failure upstream, possibly after a brief attempt at local recovery. For example, while the end host might be willing to wait indefinitely for a broken link to be restored, an intermediate host may prefer to re-attempt a connection for some finite amount of time before giving up and cascading the failure backwards. Each intermediate host along the way may make its own attempt at recovery (choosing a different path, perhaps) before, in turn, passing the failure backwards. Eventually, the origin host receives the

failure and makes the final decision.

Because end hosts at least partially share the fate of the intermediate hosts, we do not always expect an end host to be able to recover from a failure at an intermediate host. Some situations may be fatal, such as when an intermediate host knows the next hop is permanently unreachable.

### **Data at Intermediate Hosts**

Intermediate hosts should only provide services that support communication in intermittently-connected networks. In particular, they should not modify application-level data and their services should be idempotent and deterministic. That is, if the same data reaches an intermediate host more than once, the data that the host passes on to the next hop should always be the same. The services described in Chapter 2 (rendezvous points, data forwarding etc.) are examples of services that meet these requirements, while a service like stateful packet inspection and modification is not.

## **3.2 Implementation**

We have implemented a prototype of the *Euonym* architecture in Java 1.5 and use it to demonstrate the architecture using both *Euonym*-enabled applications as well as regular applications that interface with *Euonym* through proxies. In this section we describe our example implementation. Results from using it are presented in Chapter 4.

### **3.2.1 API**

*Euonym* provides the application with a socket-like, stream-oriented abstraction of the network. The API is similar to the Java Socket API with some additional support for intermediate hosts. We have developed two transport-layer interfaces, one that uses TCP sockets and one that supports file-based transfer. The latter is suitable for use with any batch transport such as data couriers or the

postal system.

The two primary classes are `NamedSocket` and `NamedServerSocket`, which are analogous to the Java `Socket` and `ServerSocket` classes. They tie in closely with the TCP interface which is described in Section 3.2.2. Tables 3.2 and 3.3 show basic use of these two classes.

### Outgoing Flows

The `NamedSocket` class acts as the outgoing half of a Euonym host, at both the origin host and at intermediate hosts. To establish a flow, an application creates an instance of the class and calls the `connect()` method, giving it a destination name or name stack to connect to. The class does the work of resolving the name, establishing a TCP connection, negotiating name stacks, etc. It provides an `OutputStream` object that the application uses to write data to the flow. The application signals the end of the flow either by closing the stream or by calling the `close()` method on the socket. The class then signals the end of the flow to downstream hosts and terminates the transport-level connection.

The same class also aids the application by listening for an incoming flow from the destination host that appears to be the mirror of the outgoing flow. That is, it watches for flows that originate at the outgoing flow's final destination host and have a final destination of the local host. The incoming flow is not automatically created and must be established independently of the outgoing flow by the remote host. The class provides an `InputStream` object that the application can read incoming data from.

### Incoming Flows

The `NamedServerSocket` class comprises the incoming half of a host and is used at intermediate hosts and final end hosts. The class operates in much the same way as a regular `ServerSocket`. The application creates an instance of the class, binds it to a name and, because our implementation is TCP-oriented, provides a port number on which to listen for incoming connections. The application

```
import euonym.net.*;
import euonym.naming.*;
import java.io.*;

public class Client {
    public static void main(String[] args) {
        try {
            NamedSocket socket = new NamedSocket();

            // name stack, top on left
            Names remote = Names.parseString(
                "a000000000000000 b000000000000000 1000000010000000");

            // connect to remote host on port 1234
            socket.connect(new NamedSocketAddress(remote, 1234));

            // send a single line
            socket.getOutputStream().write("hello\n".getBytes());

            socket.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Table 3.2: Example client code that uses a name stack to provide source-specified routing.

```
import java.io.*;
import euonym.net.*;

public class Server {
    public static void main(String[] args) {
        try {
            // bind to localhost (1000000010000000) on port 1234
            NamedServerSocket server = new NamedServerSocket(1234);

            // wait for a single incoming flow
            NamedSocket socket = (NamedSocket) server.accept();
            server.close();

            // read a line and print it to the screen
            BufferedReader r = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            System.out.println(r.readLine());

            socket.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Table 3.3: Example server code that waits for incoming flows destined for 1000000010000000.

---

then calls `accept()` which blocks until an incoming flow is available. When a new incoming flow arrives, a corresponding `NamedSocket` object is created and passed up to the application via `accept()`. The server socket multiplexes new incoming transport connections such that if they correspond to an existing flow, they are added to it, otherwise a new `NamedSocket` object is created and returned by `accept()`.

As before, the new `NamedSocket` object passes incoming data up to the application through an `InputStream` object. It also helps the application by automating the creation of the mirror, outgoing flow if needed. When the application writes to the provided `OutputStream` object for the first time, the outgoing flow is created with the local host as the origin and the origin host of the incoming flow as the destination. If the application never writes to the socket, the outgoing flow is never created.

When the application closes the `NamedSocket`, the transport connection for the incoming flow is also closed and any subsequent connections are ignored. The incoming flow itself is not actually closed: since it is one-way only the originating host can close it. If the origin continues sending data, any intermediate hosts will eventually stop trying to connect and pass the error back to the origin where it causes an exception.

### 3.2.2 TCP Interface

Our implementation uses TCP as its primary transport mechanism and provides basic but effective support for connection resumption across disconnections and address changes. An overview of the interaction between TCP and Euonym is shown in Figure 3.5.

When establishing a flow, the host resolves the destination name stack until it finds an IP address. The address can include a port number so that a name can identify a service on a host (names cannot include port numbers). A TCP connection is established with the next hop, the relevant stacks and names are exchanged, and flow data is sent. Because flows are one-way, data is never read

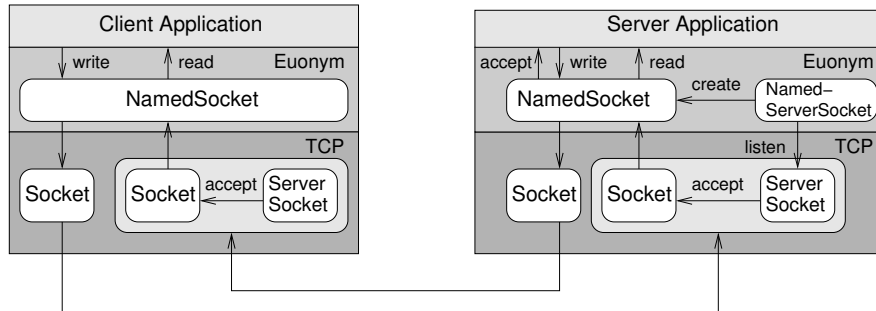


Figure 3.5: Overview of the Euonym API and its interaction with TCP. The client uses a `NamedSocket` object as the endpoint for an outgoing flow and the mirror incoming flow. The outgoing flow uses a client TCP socket to connect to the remote host and a listening TCP server socket accepts connections for the incoming flow. The server uses a `NamedServerSocket` to listen for incoming flows and create corresponding `NamedSocket` objects.

from the next hop—which corresponds to a separate flow—using the same TCP connection. That is, after the handshake, only one half of the connection is used. In a full implementation this could be optimised by using both halves if two different flows happen to use the same two intermediate hosts, but in opposite directions.

To support the return flow, if any, an outgoing flow causes a listening TCP socket to be opened. The socket waits for incoming connections that appear to be the mirror of the outgoing flow. When one arrives, any existing incoming connection for the flow is terminated and data from the new connection is passed up to the application when it reads from the `NamedSocket`'s `InputStream` object.

### Disconnection Tolerance

The implementation hides non-fatal TCP failures from the application. If a host is unreachable, it waits for an application-controlled period (5 seconds by default) before re-resolving the name and attempting the connection again. The application has control over how many times this cycle is repeated before

---

an exception is raised, possibly less for intermediate hosts than for end hosts, which may try indefinitely. If an outgoing connection has been terminated then subsequent writes to the flow block until a new connection is established. Conversely, if an incoming connection has been lost then subsequent reads block until a new incoming connection for the incoming flow is established.

Failures related to the hosts rather than the network—such as rejection of a connection by the remote host or failure to bind to a local port—cause exceptions immediately.

### **Byte Numbering**

*Euonym* hosts use byte numbering to identify their position in a flow's data stream and to prevent out of order or duplicate data. When a connection is established or resumed, the two hosts must first agree on their positions in the stream. Since a flow is one-way, there is always a sending host and a receiving host. The sending host buffers the most recent  $x$  bytes to be written to the flow and informs the receiving host of the range of bytes it is able to provide. The receiving host keeps track of the last byte number that was read by its application. If the host is the flow's final endpoint it knows exactly what byte should be delivered to its application next. An intermediate host is generally not so knowledgeable and will request as much data as possible from its upstream host, in an effort to meet the potential requirements of the next hop. If possible, the two agree on a byte to resume from and the data can begin to flow. Otherwise the receiving host rejects the sending host's connection. Like any other failure, this may be cascaded back to the start of the flow.

The remaining issue is how much data a host must buffer—the value of  $x$ . TCP must solve the same problem, but because of the connectivity assumptions of the Internet, it can place a reasonably small upper bound on the amount of unacknowledged data it sends. Our architecture does not have this luxury and passes the decision on to the application. Conceptually, it depends on how far back in the “past” the application is willing to go to get the data it must re-send.

---

For a file transfer, this may simply be the start of the file. For an interactive application, it may only be a few minutes worth of buffered interaction. Unlike TCP which hides buffered in-flight data from the application, Euonym gives the application full control over the buffering process. By default, the most recent 128KB of data are buffered. We have found this to be large enough to cover unacknowledged TCP data as well as data buffered on the local and remote sockets but not yet read by the application.

### 3.2.3 File Interface

We have also implemented a file interface that is used for preparing flow data for batch transport. For an incoming flow, the origin name, the name stack, and byte number are written to a file, followed by the flow data. The file can then be transported by any means to another host where it is injected back into the network. For an outgoing flow, the name stack and byte number are read from the file and used to establish the flow. The data is then read and streamed into the flow.

### 3.2.4 Names: Format, Storage and Lookup

Names in our implementation are 64-bit random numbers, displayed in hexadecimal. We have a single special name, 1000000010000000, which always corresponds to the local host and is analogous to the IP address 127.0.0.1. Having this address makes using Euonym on a single host much simpler.

#### Name Storage

Because Euonym names are non-aggregable, we cannot use an existing name storage system such as the Domain Name System (DNS). Instead, Euonym uses FreePastry 1.4 [9], an implementation of the Pastry [22] Distributed Hash Table (DHT) to store and lookup name mappings. A complete discussion of FreePastry and DHTs is beyond the scope of this thesis and we refer the interested reader to [22] and related work. Briefly, they work as follows. Every node in the network

has a name in a  $b$ -bit wide circular address space (a common value for  $b$  is 128), generally chosen randomly. An item to be stored in the hash table is given a identifier  $i$  taken from the same address space. The item is then stored at the node with name  $n$  such that some distance metric between  $n$  and  $i$  is minimal for all the nodes currently in the network. DHTs are naturally suitable to flat identifiers and Pastry provides scalable, efficient lookups and is resilient to node failures.

Each Euonym host runs a name server process which performs name resolution for the host and acts as a node in the Pastry DHT. Together, these hosts form a Pastry overlay and cooperate in the storage and lookup of names. A host's Pastry ID is unrelated to its Euonym names and is chosen randomly. Euonym names are mapped to Pastry's 128-bit IDs simply by padding the name with zeros. Name mappings are stored redundantly at  $k$  nodes in the overlay to help prevent losses when the overlay is partitioned or a node leaves, and Pastry ensures that mappings are re-distributed as nodes join and leave the network. In a full implementation, a host could periodically do a lookup of its own names and re-insert the bindings if it found they were missing or out of date. Additionally, a host might register a name callback with the DHT node that stores the name and be notified when the name's mapping changes.

### Regions

Intermittent and disconnected links form natural borders around well-connected hosts and divide the larger network into separate regions. Because Pastry is reliant on a well-connected TCP network, the result is a single DHT per region. This allows a name to have different mappings in different networks which is necessary for late binding and re-binding (Section 3.1.6). Conversely, if two networks are sufficiently well-connected to form a single DHT, they are sufficiently well-connected to be considered a single network.

Gateway nodes that span regions can participate in each region's DHT in two ways. Either they can run separate DHT node applications, each bound to the

---

network interface for that region, or, if they span only two regions (a common case), they can participate only in the region in which they make lookups. For example, a host that unidirectionally bridges region  $R$  to region  $S$  need only participate in the latter's DHT, provided its name resolves to its address in  $R$ 's DHT.

### 3.2.5 Limitations

Providing disconnection tolerance using TCP is more complex than it should be, mainly because TCP sockets do not provide much information for recovering from a connection error. For example, an application could make a better guess at what data to resend to a host if it knew the last byte that it acknowledged. Additionally, Java's socket implementation does not provide useful information about why a disconnection occurred or why connection setup failed, even if the underlying system provides it. All disconnections raise a `SocketException` exception and any connection setup error raises a `ConnectException` exception. Both may be raised for a number of reasons.

Completely out-of-order data delivery is not supported by our implementation. It would be useful to allow a host to accept any and all data from multiple upstream hosts and piece it together as necessary, passing the resulting contiguous stream up to the application. This could be a service performed by an intermediate host, particularly if there is a point in the network where multiple data paths converge.

## 3.3 Summary

In this chapter we described the Euonym architecture and our example implementation.

The Euonym architecture uses a layer of flat names above IP addresses to identify hosts and services in the network. Through name stacks, which contain both names and transport-layer addresses, it allows intermediate hosts

---

to be explicitly interposed on a flow between two endpoints. Name stacks also provide late binding and source- and destination-specified routing. The flexible interpretation of names allows them to identify hosts, services, networks and network paths, depending on where and how they are used.

Our prototype implementation of the architecture lies between the application and the transport layer and provides a socket-based stream abstraction of the network. It uses TCP as its primary transport protocol and tolerates network address changes and disconnections of arbitrary length. We use the Pastry Distributed Hash Table (DHT) to provide scalable, robust lookup of names in the network. Separating DHTs based on connectivity boundaries partitions the network into regions, allowing a name to have different mappings in different parts of the network.

In the next chapter we demonstrate the use of the implementation.

## Chapter 4

# Using Euonym

In this chapter we demonstrate the use of the Euonym implementation. Our experiments illustrate its applicability to both batch and round-trip dependent communication, as well as with legacy and Euonym-enabled applications. We show that it meets the architectural goals identified in Section 2.1 and, in doing so, the research goals in Chapter 1.

We performed these experiments by running all programs on the same host and simulating link availability. Unmodified applications work with the architecture through proxies and link speeds are simulated by limiting the rate at which applications can write data to their sockets. Each separate network region has its own DHT and hence naming domain.

### 4.1 Batch Communication

In this experiment we transfer 116 340KB of data across two intermittent links using the support of intermediate hosts. The experiment demonstrates use of the architecture by unmodified applications, tolerance of disconnections that exceed the limitations of the transport layer (TCP), interposition of arbitrary intermediate hosts mid-flow, and the use of intermediaries for data forwarding and to bridge heterogeneous network elements.

We model our experiment after the community example from Section 2.3. The network and link availability is shown in Figure 4.1. Two isolated community networks,  $G$  and  $H$ , are connected to the Internet ( $I$ ) by 16KBps intermittent links.  $G$ 's link is up for 20 minutes every 30 minutes and  $H$ 's link for 20 minutes every 40 minutes. Host  $A$  in  $G$  uses `scp` to copy a file from host  $B$  in

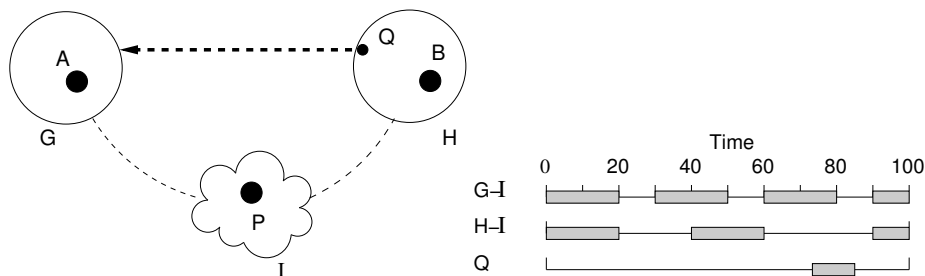


Figure 4.1: Network layout and link availability for the batch experiment.  $G-I$  and  $H-I$  are 16KBps intermittent links,  $Q$  is a high-bandwidth, high-delay data courier.

$H$  using host  $P$  in the Internet as a forwarding point. Part of the transfer is performed by a high-bandwidth data courier moving directly from  $H$  to  $G$ .

The name mappings are shown in Table 4.1. The lack of name stacks in the mappings is deceptive; they are built and consumed as data moves through the network, rather than being explicitly included from the start. For instance, at  $A$  the name  $B$  resolves to  $\{p, P, H, B\}$  and when  $B$  invokes  $Q$ , it resolves the name  $A$  to  $\{q, Q, G, A\}$ .

The current version of the `ssh` and `scp` protocols (version 2 [32]) rate limit its connections and requires application-level acknowledgement of a window of data (by default 16KB) before further data is sent. This limits the ability of forwarding points to buffer data when the end host is unavailable. Instead, we use version 1 of the protocol [31] which does not use rate limiting. Another approach would be to use a sufficiently large window, although deciding what constitutes “sufficiently large” is difficult.

The flow of data during the experiment is shown in Figure 4.2, which should be read from the bottom upwards. Initially, both links are up and after the initial `scp` handshake data flows from  $B$  to  $A$ , via  $P$ , at 16KBps. After 20 minutes, both links go down. At 30 minutes, the  $G-I$  link comes up again and remains idle until the  $H-I$  link comes up. When the  $G-I$  link goes down again at 50 minutes, data from  $B$  continues and is buffered at  $P$ . At 60 minutes,

Mappings in $G$	Mappings in $H$	Mappings in $I$
$A \Rightarrow \{a\}$	$A \Rightarrow \{G\}$	$A \Rightarrow \{a\}$
$B \Rightarrow \{H\}$	$B \Rightarrow \{b\}$	$B \Rightarrow \{b\}$
$G \Rightarrow \{\}$	$G \Rightarrow \{P\}$	$P \Rightarrow \{p\}$
$H \Rightarrow \{P\}$	$H \Rightarrow \{\}$	
$P \Rightarrow \{p\}$	$P \Rightarrow \{p\}$	
	$Q \Rightarrow \{q\}$	

Table 4.1: Name mappings for the batch experiment, with transport-level addresses in lowercase. When invoking  $Q$  in  $H$ , the mapping for  $G$  becomes  $G \Rightarrow \{Q\}$ .

when  $H-I$  goes down and  $G-I$  comes up, the data buffered at  $P$  is sent to  $A$  for the next 10 minutes. We deliberately prevent the  $H-I$  link from coming up at 80 minutes. Instead, at 74 minutes another intermediary  $Q$  in network  $H$  is invoked that reads 71 677KB of data from  $B$  at 512KBps and writes it to a file. A few minutes later, the data is injected back into network  $G$ . Finally, both links are up again at 90 minutes and the file transfer is completed at 98 minutes.

The transfer averages a rate of 19.8KBps over the entire 98 minutes. The two links are up simultaneously for only 40 minutes, and at 16KBps would regularly only be able to transfer 38 400KB (assuming that they resume the file transfer mid-way—not possible using scp—and ignoring the connection setup overhead).

## 4.2 Interactive Communication

In this experiment we retrieve a collection of webpages across a series of intermittent links. It demonstrates the use of round-trip dependent communication in the architecture, support for communication without a connected end-to-end path, and asymmetrical flows.

The experiment is loosely modelled on the Wizzy project described in Sec-

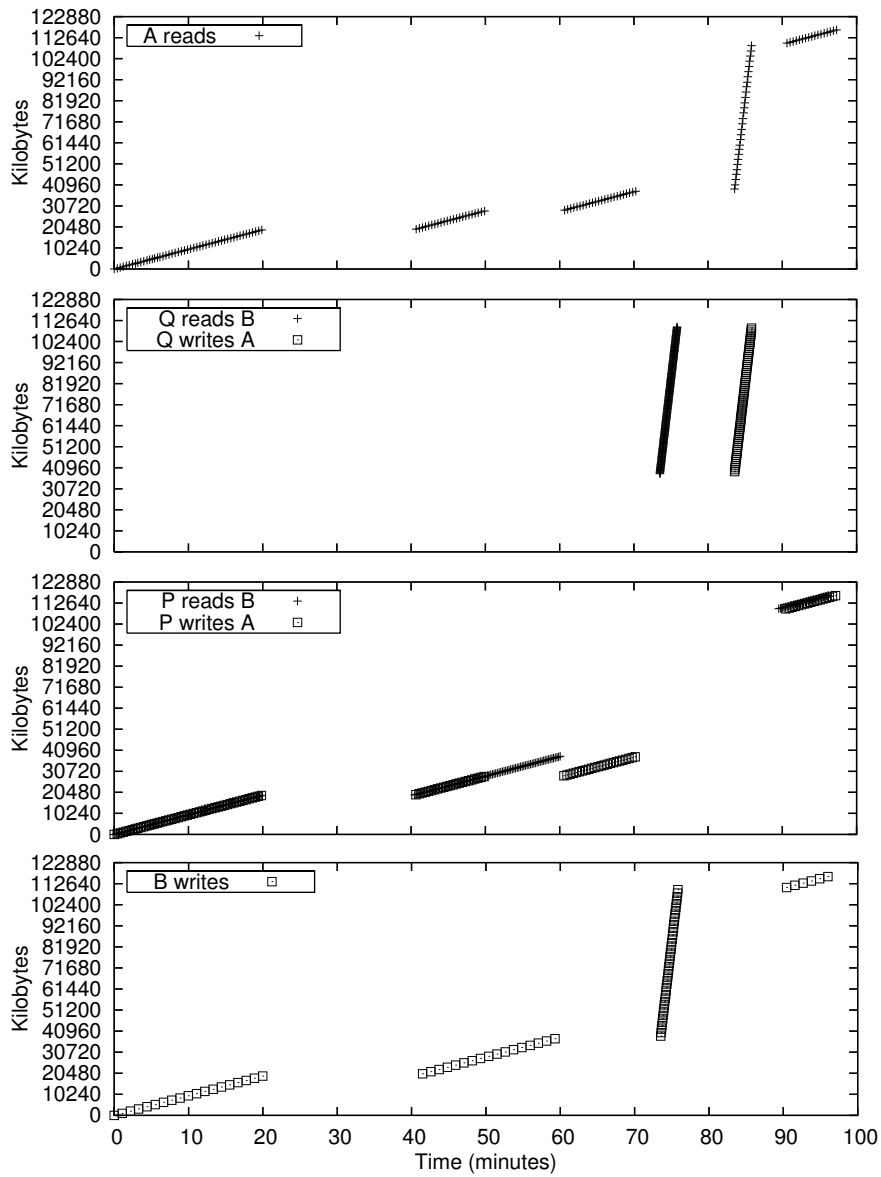


Figure 4.2: Flow of data from *B* to *A* in the batch experiment. The figure can be read from the bottom upwards, following the data from *B* to *P* (or *Q*) and finally to *A*. Data is sent in the opposite direction only at connection setup and teardown and is not shown.

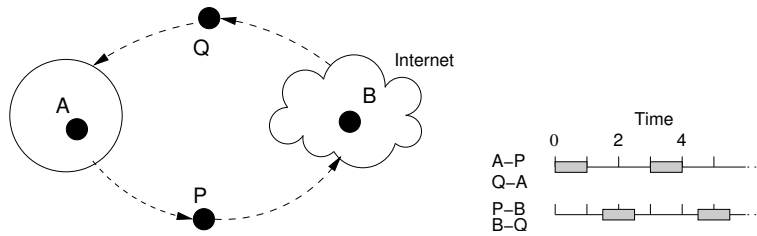


Figure 4.3: Network layout and link availability. Requests are made by  $A$  and passed onto the Internet by a proxy at  $B$ .

tion 2.3, the scenario being a student wishing to use Google to find information on a topic without requiring an internet connection. We use the Google API to search for the phrase “south africa” on `en.wikipedia.org` and the top three results are retrieved, along with any embedded images, using the `wget` tool. The transfers are performed across a series of intermittent connections and the student can then browse the results locally, offline.

The network setup is shown in Figure 4.3. Host  $A$  issues all requests, including the Google search which uses HTTP-based RPC, through a simple Euonym-enabled HTTP proxy at the Internet host  $B$ . The requests go from  $A$  to  $B$  via host  $P$  and the responses from  $B$  to  $A$  via host  $Q$ . Non-internet links are limited to 16KBps. Links are up for 1 minute and down for 2 minutes, with the  $Q$ - $A$  and  $A$ - $P$  links starting with their up cycle and the  $P$ - $B$  and  $B$ - $Q$  links with their down cycle. Hence, there is never a fully connected path between  $A$  and  $B$  in either direction. From  $A$ 's perspective,  $B$  maps to  $\{P, B\}$  and from  $B$ 's perspective,  $A$  maps to  $\{Q, A\}$ . Otherwise, all host names map to their transport addresses.

The flow of data is shown in Figure 4.4. The first request (at time 0) is the Google search and the result arrives at  $A$  just after 3 minutes. The next three requests are for the HTML content of the webpages and have reasonably large responses. The subsequent requests are for the images in the webpages. Every round trip takes approximately 3 minutes and, because `wget` waits for one item to be retrieved before requesting the next, progress is slow. Clearly, parallelising

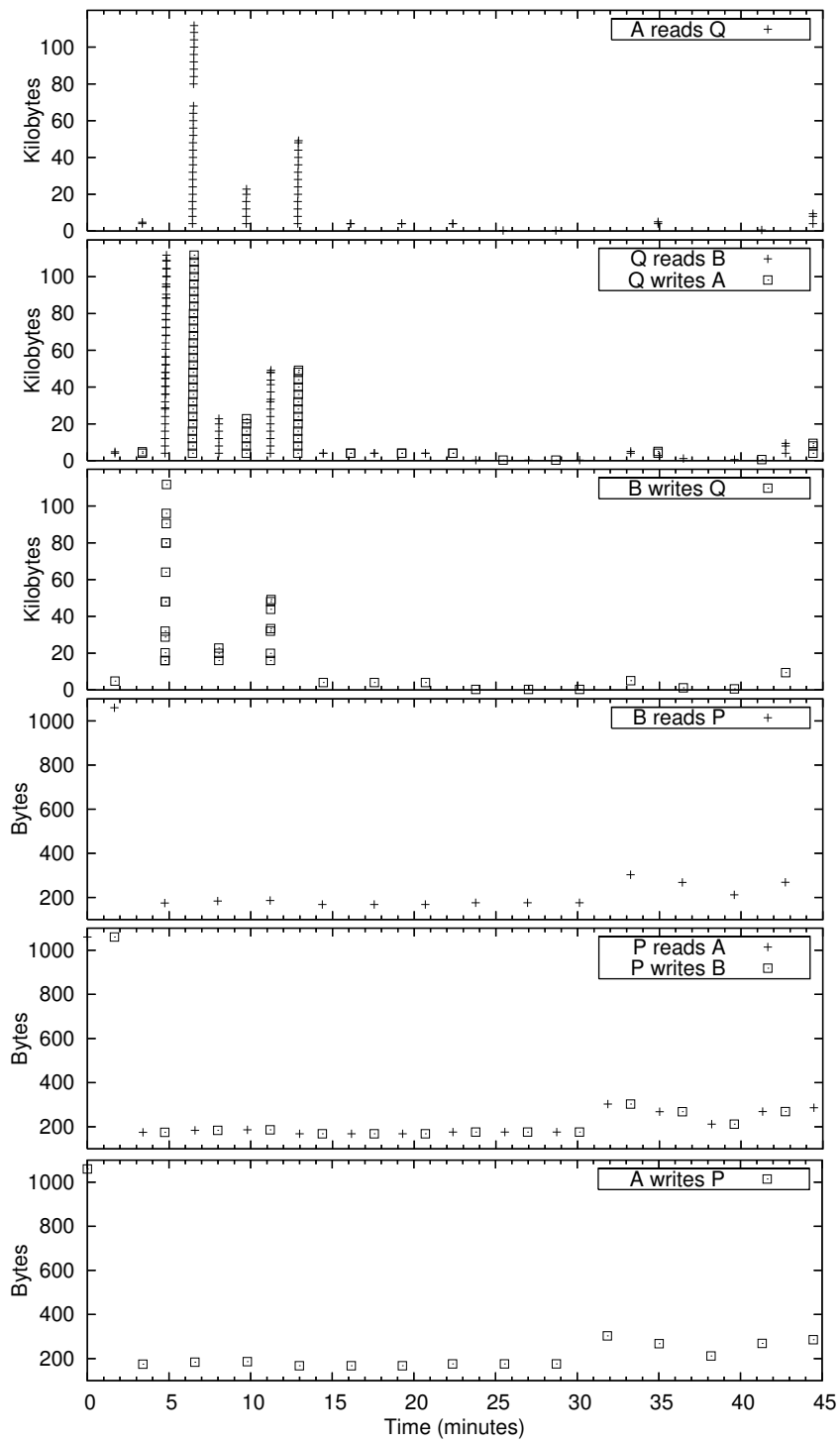


Figure 4.4: Flow of data in the interactive experiment. Reading upwards, a request travels from *A* to *P* to *B* and the response from *B* to *Q* to *A*.

the retrieval and reducing round trip dependence by moving the program issuing requests closer to  $B$  would improve performance hugely. Nevertheless, the pages and their images are successfully retrieved despite no connected end-to-end path and multiple disconnections.

A total of 217KB (or 15 requests) are downloaded in 47 minutes, an average rate of 79bps. This is significantly lower than the links' 16KBps limit because of the dependence on lengthy, serial round trips. However, even this speed is an improvement over ordinary circumstances in which no transfer at all is possible due to the lack of an end-to-end path.

### 4.3 Remote Service

This experiment is similar to the previous one but pushes the request logic into the Internet itself, reducing round trip dependence and improving performance. It shows the use of explicit source-specified routing and both the client and server are completely Euonym-enabled and do not require proxies. It also illustrates how the location of a service can affect its performance.

The network setup and link availability are the same as for the previous experiment. As before, we use the Google API to download the webpages and related images for the first three results in a search for the phrase "south africa" on `en.wikipedia.org`. In this experiment, though, the client at  $A$  sends only the search string and the required number of results to the server at  $B$ .  $B$  performs the search, retrieves the results using `wget`, and sends the downloaded data back to  $A$ . We therefore reduce the communication to one round trip and make much better use of the available bandwidth.  $A$  uses source-specified routing to contact  $B$  by specifying  $\{P, B\}$  as the destination name stack. All other name mappings are as before.

The flow of data is shown in Figure 4.5. At time 0,  $A$  sends its request (via  $P$ ) which reaches  $B$  at 100 seconds. At 120 seconds, after performing the search and web retrievals,  $B$  begins sending the data back to  $A$ . It is buffered at  $Q$  until the  $Q$ - $A$  link comes up at 200 seconds. The transfer of 330KB is

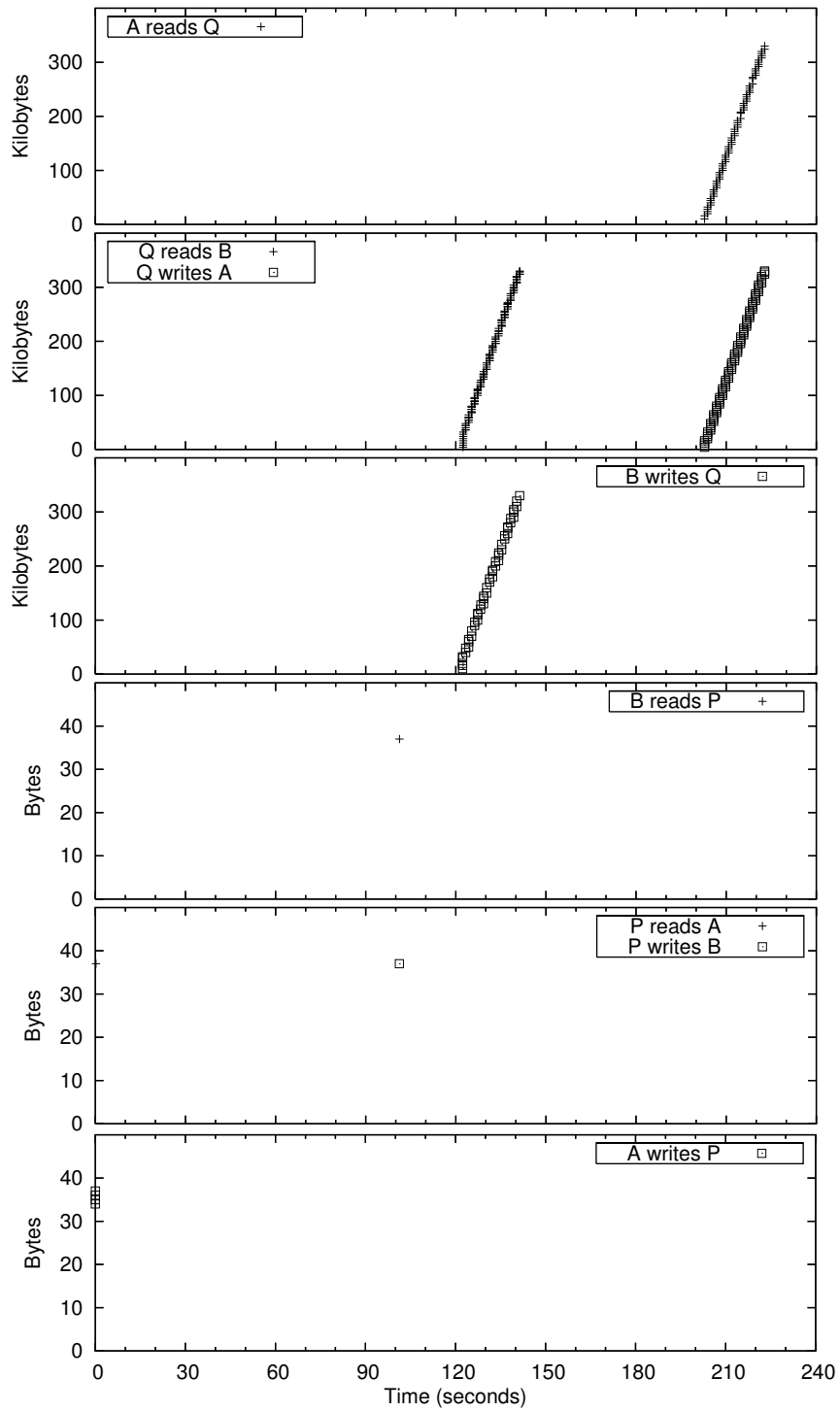


Figure 4.5: Flow of data in the remote service experiment. Reading upwards, the request travels from *A* to *P* to *B* and the response from *B* to *Q* to *A*.

complete after 223 seconds. A change in the top 3 search results increased the amount of data transferred, but an average throughput of 1.4KBps and an order of magnitude improvement in transfer time are great improvements over the previous experiment. Again, with no end-to-end path, no transfer would normally be possible at all.

Note that we could couple the retrieval service at *B* with the HTTP proxy from the previous experiment and move the service anywhere between the client and the Internet. Clearly, the fewer round-trip dependencies the better.

## 4.4 Additional Discussion

There are two additional aspects of Euonym that do not lend themselves to network traces or bandwidth measurements and are instead simply described below. The first is mobility and the second is support for on-the-fly changes in communication paradigm.

### 4.4.1 Mobility

Mobility support is inherent in the Euonym architecture and implementation. Flows are established between hosts themselves, not network locations, and arbitrary-length disconnections are tolerated by default. We have used Euonym to preserve ssh connections from a mobile laptop to a server for weeks at a time. Euonym successfully handles multiple address changes, NAT devices and firewalls, NIC changes, periodic network disconnections<sup>1</sup>, and system suspension (hibernation). Furthermore, because ssh provides anti-replay security, there are no dangers of man-in-the-middle attacks or in resuming connections while on untrusted networks.

From a personal standpoint, once one becomes used to ones connections withstanding both minor and major interruptions, it is frustrating to go back to the original situation, especially when operating in a mobile environment.

---

<sup>1</sup>Such as those brief but frequent interruptions that seem to plague wireless networks.

---

We long for the day when support for mobility and disconnection tolerance is widespread.

#### 4.4.2 Switching to a Rendezvous Paradigm

We have also used Euonym to allow a transition from a regular, *sender/recipient* communication paradigm to a rendezvous-style *put/get* paradigm. During an established interactive ssh session, we adjust names to introduce two new servers into the client-server flow. These two servers form a rendezvous pair: the downstream host connects backwards to the upstream host to ask for data, rather than the reverse. The downstream host then uses the associated name stack to continue sending the data along the flow. The setup at the end hosts is identical to that in the interactive communication experiment in Section 4.2.

This example illustrates how even a fundamental communication paradigm change can be handled easily by the architecture simply by pushing the logic into the intermediate hosts. Such a paradigm is useful when the upstream host is unable to contact the downstream host directly, possibly because ingress connections cannot be established directly (as in the case of a blocking firewall between the hosts).

### 4.5 Summary

In this chapter we present experiments performed using Euonym to demonstrate the feasibility of the architecture. The first experiment demonstrates batch-style communication using legacy applications, communication in the face of disconnections and the use of atypical transport mechanisms to improve connectivity. The second shows round-trip dependent communication using a mixture of legacy and Euonym-enabled applications and demonstrates communication without a fully connected end-to-end path. The third experiment uses only Euonym-enabled applications and shows how the location of a service in the network can affect its performance. We also discuss how Euonym provides simple

but effective mobility support and allows an ad-hoc change to a rendezvous-style communication paradigm.

The experiments and usage experience show that the architecture meets the goals set out in Chapter 1. To wit, we have shown that

1. explicitly including proxies and other intermediary hosts in a path between two endpoints facilitates communication in intermittently-connected networks; and
2. the separation of network address and host identity allows end hosts to delegate responsibility and operations to intermediary hosts that are better equipped to make use of, or improve upon, connectivity.

In the next chapter we discuss related work.

## Chapter 5

# Related Work

In this, the penultimate chapter of the thesis, we discuss related work. We begin with work related to disconnection tolerance, then cover work on intermittently-connected networks, and finally work on host identity and addressing.

### 5.1 Disconnection Tolerance

With the abundance of connectivity we have today, few new applications are designed with disconnection in mind. However, a variety of projects have focused on tolerating disconnections at the transport layer, generally from a mobility perspective. Some lie between the application and the transport and let the application take control of mobility and disconnections, leaving the lower layers intact. Others modify the network or transport layers and attempt to hide disconnections from the application entirely.

Zhang and Dao [34] propose a *persistent connection* model for continuing connections across process migration and host crashes. They place a session layer above the transport layer and separate identity from network address by using a portion of the IP address space to assign identifiers to processes (rather than hosts). Their focus is on host failure and so they do not address the possibility of losing data in the network, which Euonym must consider.

Zandy and Miller [33] describe *rocks and racks*, reliable sockets and packets that support connection mobility and disconnections of arbitrary (but pre-specified) length through a user-level library. It is aimed at allowing existing applications to work unmodified and hides disconnections entirely. Endpoints

negotiate support for enhanced sockets during TCP connection setup<sup>1</sup> and perform a key exchange to identify each other later.

Snoeren and Balakrishnan [26] advocate an approach where the host takes responsibility for mobility, rather than the network. They add a new TCP *migrate* option, negotiated at connection setup along with a key exchange, and a corresponding state to the TCP state transition diagram. They use DNS to map between host identity and address and the focus is on mobility, rather than disconnection tolerance: only disconnections within the TCP timeout period are tolerated.

The *robust TCP connections* described in [4] are a session layer approach that tolerates arbitrary-length disconnections. Reconnecting a broken connection involves re-negotiating byte numbers and must be initiated by the client host. They use UDP to send out-of-band control packets to avoid the problem of embedding control data in the application data flow. The concept is similar to that used by Euonym, although we use in-band control data and whichever host writes to a flow can trigger a reconnection.

Maintaining connections across address changes in support of mobility, if not specifically disconnections, has been proposed at a much lower level in both IPv4 [20] and IPv6 [14]. Long-lived disconnections are not supported and connections are resumed based on the ability of a host at a new address to affirm that it is indeed the “old” host. That is, the end-to-end connections are between addresses, not persistent host identifiers; there is no separation of host identity and network address.

## 5.2 Intermittently-Connected Networks

The Delay Tolerant Network [5, 6] architecture bridges heterogeneous and intermittently connected networks using a store-and-forward model with a *bundle*, rather than a stream, abstraction. Bundles are forwarded by agents placed

---

<sup>1</sup>The negotiation involves a certain amount of “trickery” to make it backwards-compatible, including open a socket and then closing it immediately.

---

at network boundaries that are intended to make routing decisions based on present and upcoming connectivity and are also used to bridge heterogeneous link layers. They use a two-level  $\{name, region\}$  naming hierarchy to achieve late binding and guide the bundle through the network. This is a more specific approach than we take with Euonym, in that our intermediate hosts can provide arbitrary support services, not just hop-by-hop forwarding in the greater network. In addition, we avoid the complexity of a name and region hierarchy.

A potential function of intermediate hosts is routing, an entire area of research in its own right. The problem is interesting in that network connectivity is time-varying but may be known *a priori*. Jain *et al.* [13] formulate the problem in detail and discuss possible approaches. An example of a protocol targeted at intermittently-connected networks is PRoPHET [17, 16], which performs routing based on the probability of a node encountering other nodes, either due to mobility or intermittent connectivity. An evaluation of different flooding protocols and their use in these networks is given in [10].

Wang *et al.* [29] motivate the use of the postal system as a digital communication mechanism. Arguing that it is widespread, well understood and reaches even extremely remote areas, they suggest using it to supplement or provide connectivity to remote and low-income areas, noting that it can provide very high bandwidth with reasonable delay. Euonym provides simple, effective support for such a system by using intermediate hosts as proxies to the postal network, relieving the applications of the burden of supporting such an atypical communication mechanism.

The Wizzy Digital Courier project [30] provides Internet access to isolated schools in South Africa. They use application-specific methods to delay web queries, emails and other traffic until an Internet connection is available. This connection is made either during off-peak hours when a dialup connection is cost-effective, or by transporting the cached queries on a USB drive to an Internet-enabled host, waiting for the replies, and then transporting them back.

The Sámi Network Connectivity (SNC) project [23] provides connectivity to isolated communities of reindeer herders in northern Scandinavia. The commu-

nities are nomadic and intermittently connected to each other and the outside world by satellite, long-range radio and the postal service.

The Euonym architecture has direct applications in both these environments, our aim being to make these types of project much easier to implement.

### 5.3 Identity and Addressing

The following projects all address the need to separate host identity from network address and location.

Balakrishnan *et al.* [1] draw on the literature to argue for an additional naming layer above the existing IP layer. They argue that it will help resolve mobility and multi-homing issues in the current Internet and mitigate the effects of middleboxes by making them first-class citizens in the network. In addition, they suggest using opaque, non-aggregable, globally-unique identifiers not just to name hosts but also objects and data in the network.

The Host Identity Protocol [18, 11] uses cryptographic key-based hostnames to separate network address and identity. The architecture lies between the transport and the IP layers and as a result transport protocols are bound to endpoint names, rather than addresses. The use of both the DNS and DHTs as name storage mechanics is being investigated. The architecture is in the early stages yet and the infrastructure required to support experimental use of it is under development. Shütz *et al.* [25] describe a modification to TCP to support user-specified arbitrary-length disconnections and mobility using HIP.

The Unmanaged Internet Protocol [7] describes an identity-based naming and routing overlay that allows groups of devices to form unmanaged, ad-hoc networks that operate equally well on the periphery of larger networks, as well as in isolation. Hosts are identified in a manner similar to HIP and routing is DHT-based. Nodes essentially form an overlay network and cooperate to negate hassles usually associated with NATs and firewalls.

Walfish *et al.* [28] describe a delegation-oriented architecture that makes middleboxes (firewalls, NAT devices etc.) first-class network citizens through the

---

use of persistent identifiers. The architecture lies between the IP and transport layer and does not address mobility or multi-homing. In addition, an end host cannot control which intermediaries are invoked based on who is communicating with it. Because the transport protocol becomes bound to host identifiers, the behaviour of intermediate hosts is limited; they cannot terminate an incoming connection and use an entirely different one on the outgoing portion, and they are unable to span disconnections that are not tolerated by the transport protocol.

The Internet Indirection Infrastructure (*i3*) [27] uses lists of host identifiers to support multicast and anycast in a DHT-based overlay network. It provides *put/get* rendezvous-style communication to separate the sender and receiver, resulting in a form of disconnection tolerance. They use opaque, aggregable names called *triggers* to identify hosts within their overlay network. Zhuang *et al.* [35] describe a mobility system based on *i3*. End hosts are responsible for their own mobility and since they control the triggers they place in the overlay, they can control the efficiency of routing and handoff.

## Chapter 6

# Conclusion

In this chapter we conclude the thesis and discuss possible lines of future work.

The apparent ubiquity of the Internet and reliance on it for every-day work has led to an exploration of ways to provide Internet-like connectivity in areas either previously lacking it or dominated by alternative, domain-specific networks. Such a move provides many challenges, such as overcoming a reliance on reasonable-bandwidth, low-latency connectivity and connected end-to-end paths. Implicit in the design of the Internet family of protocols, these requirements are often hard to meet in the class of so-called *challenged* networks.

This thesis describes an architecture that aims to promote Internet-style connectivity in these environments by allowing intermediate helper hosts to be explicitly involved on a flow between two hosts. These intermediate hosts can be included on the fly and provide a wide range of support services. These include buffering data until it can be passed to the next downstream host, performing routing to select that host, bridging heterogeneous network elements, and acting as rendezvous points when a regular *send/receive* communication paradigm is inappropriate. Through experimental demonstration we have shown how such services promote connectivity in difficult network environments without putting a burden on the application to support multiple networking technologies, routing methods etc.

Our architecture's persistent, flat naming layer provides the flexibility of source- and destination-specified routing and can be easily used to identify hosts, networks, services and routing paths. They are a key component in providing applications with transparent tolerance of arbitrary-length disconnections.

## 6.1 Future Work

This thesis shows that intermediate hosts are a viable way of exploiting intermittent connectivity. Here we describe a few ideas for further exploration.

The use of DHTs as a storage and lookup mechanism is a relatively new development in general, and needs to be explored further. It would be useful for applications to be able to push updates to their own names to remote networks, possibly through Euonym itself. It would also be interesting to investigate how a mobile node or data mule could “advertise” its services by updating bindings in networks it arrives at in order to divert traffic to itself.

Euonym used a custom naming solution to identify hosts, mainly because of the requirement for name stacks. Investigating how an architecture like Euonym could rest upon a separate naming system like HIP—which provides robust security, disconnection tolerance, support for the legacy Internet etc.—may be fruitful to both the challenged network community as well as the naming and HIP communities.

Euonym potentially allows data to arrive out of sequence at the end host, possibly from separate sources. The end host should ideally be able to piece the data together as it arrives rather than expect it to arrive in-order. Consequently, it could be useful for an end host to request retransmission of an outstanding chunk of data if it is not received within some (possibly application-defined) time period. The request could go either to the origin host directly or to the previous upstream hop and slowly be passed back towards the origin. These operations could also be performed by an intermediate host on behalf of the end host.

Involving intermediate hosts introduces challenges in its own right, such as the understanding and handling of failures and how intermediate hosts affect end hosts. In particular, a suite of protocols for interacting and managing intermediate hosts, along the lines of the ICMP, would be useful, although it need not be part of Euonym itself. This could help routing hosts to make routing decisions, advertise link availability, request missing data, etc.

# Bibliography

- [1] BALAKRISHNAN, H., LAKSHMINARAYANAN, K., RATNASAMY, S., SHENKER, S., STOICA, I., AND WALFISH, M. A layered naming architecture for the internet. In *ACM SIGCOMM* (Sept. 2004), ACM Press, pp. 343–352.
- [2] CARPENTER, B., AND BRIM, S. Middleboxes: Taxonomy and issues. RFC 3234, Feb. 2002.
- [3] The IRTF Delay Tolerant Network Research Group (DTNRG), Aug. 2005. <http://www.dtnrg.org/>.
- [4] EKWALL, R., URBÁN, P., AND SCHIPER, A. Robust TCP connections for fault tolerant computing. In *Proc. 9th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec. 2002).
- [5] FALL, K. A delay-tolerant network architecture for challenged internets. In *ACM SIGCOMM* (2003), ACM Press, pp. 27–34.
- [6] FALL, K. Messaging in difficult environments. Intel Research Berkeley, Technical Report IRB-TR-04-019, Dec. 2004.
- [7] FORD, B. Unmanaged Internet Protocol: Taming the edge network management crisis. In *2nd ACM Workshop on Hot Topics in Networks (HotNets II)* (Nov. 2003).
- [8] FRANCIS, P., AND GUMMADI, R. IPNL: A NAT-extended internet architecture. In *ACM SIGCOMM* (Aug. 2001), ACM Press, pp. 69–80.
- [9] FreePastry 1.4, Mar. 2005. <http://freepastry.rice.edu/>.

- 
- [10] HARRAS, K., ALMEROOTH, K., AND BELDING-ROYER, E. Delay tolerant mobile networks (DTMNs): Controlled flooding schemes in sparse mobile networks. In *IFIP Networking* (May 2005).
- [11] The IETF Host Identity Protocol (HIP) Working Group, Aug. 2005. <http://hip.piuha.net/>.
- [12] The IRTF Interplanetary Internet Special Interest Group (IPNSIG), Mar. 2004. <http://www.ipnsig.org/>.
- [13] JAIN, S., FALL, K., AND PATRA, R. Routing in a delay tolerant network. In *ACM SIGCOMM* (Sept. 2004), ACM Press.
- [14] JOHNSON, D., PERKINS, C., AND ARKKO, J. Mobility Support in IPv6. RFC 3775, June 2004.
- [15] KLENSIN, J. Simple Mail Transport Protocol (SMTP). RFC 2821, Apr. 2001.
- [16] LINDGREN, A., AND DORIA, A. Probabilistic routing protocol for intermittently connected networks. Work in progress, Internet Draft, July 2005. <http://www.dtnrg.org/docs/specs/draft-lindgren-dtnrg-prophet-01.txt>.
- [17] LINDGREN, A., DORIA, A., AND SCHELÉN, O. Probabilistic routing in intermittently connected networks. In *Proceedings of the First International Workshop on Service Assurance with Partial and Intermittent Resources (SAPIR)* (Aug. 2004), pp. 239–254.
- [18] MOSKOWITZ, R., AND NIKANDER, P. Host Identity Protocol. Work in progress, Internet Draft, Aug. 2005. <http://www.ietf.org/internet-drafts/draft-ietf-hip-arch-03.txt>.
- [19] NIKANDER, P., YLITALO, Y., AND WALL, J. Integrating security, mobility, and multi-homing in a HIP way. In *NDSS '03* (Feb. 2003), pp. 87–99.
- [20] PERKINS, C. IP Mobility Support for IPv4. RFC 3344, Aug. 2002.

- 
- [21] POSTEL, J., AND REYNOLDS, J. File Transfer Protocol (FTP). RFC 2821, Oct. 1985.
- [22] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (Nov. 2001), pp. 329–350.
- [23] Sámi Network Connectivity (SNC) Project, Aug. 2005. <http://www.snc.sapmi.net/>.
- [24] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2, 4 (1984), 277–288.
- [25] SCHÜTZ, S., EGGERT, L., SCHMID, S., AND BRUNNER, M. Protocol enhancements for intermittently connected hosts. *SIGCOMM Computer Communication Review* 35, 3 (2005), 5–18.
- [26] SNOEREN, A. C., AND BALAKRISHNAN, H. An end-to-end approach to host mobility. In *ACM MOBICOM* (Aug. 2000), pp. 155–166.
- [27] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In *ACM SIGCOMM* (Aug. 2002), pp. 73–86.
- [28] WALFISH, M., STRIBLING, J., KROHN, M., BALAKRISHNAN, H., MORRIS, R., AND SHENKER, S. Middleboxes no longer considered harmful. In *USENIX OSDI* (Dec. 2004), pp. 215–230.
- [29] WANG, R. Y., SOBTI, S., GARG, N., ZISKIND, E., LAI, J., AND KRISHNAMURTHY, A. Turning the postal system into a generic digital communication mechanism. *SIGCOMM Computer Communication Review* 34, 4 (2004), 159–166.
- [30] Wizzy Digital Courier, Apr. 2005. <http://www.wizzy.org.za/>.

- 
- [31] YLONEN, T. The SSH (Secure Shell) remote login protocol. Work in progress, Internet Draft (expired), May 1995. <http://www.snailbook.com/docs/protocol-1.5.txt>.
- [32] YLONEN, T., AND LONVICK, C. SSH connection protocol. Work in progress, Internet Draft (expired), Mar. 2004. <http://www.ietf.org/internet-drafts/draft-ietf-secsh-connect-25.txt>.
- [33] ZANDY, V. C., AND MILLER, B. P. Reliable network connections. In *ACM MobiCom* (Sept. 2002), ACM Press, pp. 95–106.
- [34] ZHANG, Y., AND DAO, S. A “persistent connection” model for mobile and distributed systems. In *Proc. of the 4th International Conference on Computer Communications and Networks (ICCCN '95)* (1995), IEEE Computer Society, p. 300.
- [35] ZHUANG, S., LAI, K., STOICA, I., AND KATZ, R. Host mobility using an internet indirection infrastructure. In *USENIX MOBISYS* (May 2003).