Discourse with Disposable Computers: How and Why you will talk to your tomatoes

David Arnold, Bill Segall, Julian Boot, Andy Bond, Melfyn Lloyd, Simon Kaplan

CRC for Distributed Systems Technology (DSTC)
The University of Queensland, St Lucia, 4072, Australia
Phone: +61 7 3365 4310 Fax: +61 7 3365 4311

{arnold,bill,julian,bond,melfyn,simon}@dstc.edu.au

Abstract

Beyond ubiquitous computing, is the advent of disposable computing, occurring when the price of an embedded computer becomes insignificant compared to the cost of goods. Current software and network architectures and their associated programming paradigms will not scale to this new world. The necessity of catering for the constant change in number and type of devices of interest to a user, as well as their sheer quantity, dictates new approaches to construction of software systems based on more flexible models.

We propose that distributed event notification forms a fundamental requirement for systems of this scale, and discuss the advantages of undirected communication over current interaction models. Our experience with Elvin, a prototype notification system motivates the discussion and serves as illustration of its possibilities.

1. Introduction

We are rapidly approaching an era where most consumer products contain an embedded computer and network interface. While the availability of ubiquitously "wired" goods is currently a novelty, it will soon be not only commonplace, but all pervasive.

However, we contend that most predictions of ubiquitous computing drastically understate the number of networked devices. While it is easy to imagine networked toasters, fridges, televisions, and indeed, any already electronic device, following these will be the second wave of wired devices; the era of *disposable computing*, when the price of embedding a computer becomes insignificant compared to the cost of manufacture. Far more than ubiquitous computing, disposable computing will wreak fundamental changes in the nature of computing, allowing almost every object encountered in daily life to be "aware", to interact, and to exist in both the physical and virtual worlds.

In particular, disposable computing dictates a new approach to interaction amongst software components, and between software and human users. The issue facing software architects is how do we effectively use networked food, clothing, paper, books, people, doors, cars and roads? What communication strategies are needed? How do we manage quadrillions of devices? And how do they interact with us?

We begin by describing some properties of future networks, and a scenario that drives an analysis of requirements for computer-enabled interaction on a vast scale. A prototype system for pervasive, contingent component interaction is introduced, and discussed in light of the scenario. Some of our current endeavors indicate useful properties, and we discuss some of the many research challenges that remain the subject of future work.

2. The Wired World

For over a decade, computer scientists have predicted the integration of computers and networks with the affordances of our daily life [Wei91]. The development of hardware has today reached a point where this is technically viable, and it will shortly become financially accessible to average consumers.

The software challenges offered by the previous generation of hardware are being answered by technologies like Plug and Play and Jini, but the grand challenges of ubiquitous computing remain unanswered. As we examine interaction models for software, we consider four particular problems and their impact.

The Quadrillion Node Net

The first wave of consumer electronic devices with a network interface will extend the current global network to trillions of devices. But it is the second wave, the instrumentation of non-electronic devices, which ushers in the Quadrillion Node Net. When every book, packet, street sign, soda can and pen is active and networked, the number and diversity of devices challenge out ability to control and manage them.

Disposable Computing and Device Churn

How often do you buy a new computer? And when you do, how long does it take to get it set up the way you need it? When every manufactured product you see larger than a paper clip is a computer, how do you configure them? Rather than acquire a new computer every year, you will acquire them every minute, sometimes by the 1000. And you will throw or give away computers at the same rate (or your partner will finally leave you!). Objects with embedded computers will appear and disappear from the containing network at a frantic rate.

Security and Charging

When you throw you lunch wrapper in the trash, its computer negotiates with the trashcan to be recycled or shredded or composted. But your lunch wrapper was bought using your debit account, and the trash can wants to charge you for burdening it with non-recyclable plastic...

The possibility for eavesdropping and losing sensitive information becomes overwhelming once computers are disposable. The volume of data available about you and your life becomes absolutely staggering. How do we secure your information environment whilst retaining the availability and mobility of your data? How do we balance the benefits of availability whilst protecting against intrusion.

Context Management

Software components are remarkably good at ignoring unwanted stimulus, but people become quickly irritated by untimely information. The benefits of having the universe at your fingertips are quickly overlooked if the universe is always in your face. When you are responsible for a million interaction-rich computers, these interactions are going to need to be coordinated, filtered, and exchanged, but above all mediated automatically.

Users must be able to set policy for their interactions with the environment that includes the context, not only of themselves, but their interactions with other objects at any given time. Context management encompasses the mechanisms used to specify what is appropriate user interaction, and to automatically determine when and how it is appropriate.

A common, vital element in the solutions to these problems is the nature of communication between software components. Distributed systems currently use a variety of protocols, with a growing general reliance on an RPC-style model. However, RPC and remote method invocation are constrained to a request/reply interaction, using known interfaces types at a specific, possibly indirectly resolved, address.

But the universe of disposable computing is populated with devices whose type and identity are completely unknown to the other devices they will have to interact with. The continual churn of artifacts relevant to a task will completely overwhelm our current solutions of name servers and well-known addresses within the homogeneous IP network.

The next section introduces the scenario that the rest of the paper uses as the basis for analysis of these issues, and presentation of a possible solution.

3. Pasta, circa 2005[1]

Somewhere in Germany there is a factory that produces the little cans that canned food goes into. This factory makes cans that appear perfectly normal it's just that each can contains a tiny computer, a small amount of memory, and a short-range radio transceiver. It's a smart can and the factory that makes them charges eight pfennigs more for each one. As part of their production, the cans get embedded with a small amount of data such as the date of manufacture, the batch and can number, the alloy details etc.

Once produced these cans travel all over Europe. One batch of these cans is sent to Italy where they go to a tomato-canning factory and are filled with tomatoes. At this factory, as part of the canning process, the can gathers a little more data: it is full of diced Roma tomatoes, it was filled on a certain date as part of a particular batch, and it has a particular use-by date.

One of these cans of tomatoes gets exported to the USA. As it moves off the wharf it is processed and its data content is translated from Italian to English. After a brief stint in a warehouse it ends up on a supermarket shelf. At the supermarket it inherits a little more information such as the retail price and date of being placed on the shelf. At some point a customer's pantry knows to order the can and one is sent to your house in the next delivery. Before the can leaves the store, the supermarket extracts the information it needs for stocktaking.

[1] Inspired by Hiro's pizza box in Stephenson's *Snow Crash* [Ste92].

Some weeks later you're at your desk at work thinking about dinner, and decide that tonight you're going to cook a romantic meal for two. You look up your recipes, select one, and check your pantry for the necessary ingredients. Your tomatoes have cheerfully registered themselves to the pantry upon arrival, so it is able to report that all you need is some fresh basil that you can pick up on the way home.

At the supermarket, you find the basil and drop it into the trolley, which updates the cumulative price of your selections. Noticing the screen's flicker, you glance down and see an advertisement for a special on oregano. You cancel it and disable further advertising.

Finally done, you push the trolley through the checkout, where your account is debited for the total, and your home address attached to your items. You push the trolley onto the track for delivery before heading to the cafe for a coffee on the way home as the store delivers the shopping for you.

At home you begin to cook, placing the opened can of tomatoes from the pantry onto the table. The can reports that it has been opened (after detecting the pressure differential).

You've been meaning to get the auto-light on your gas stove fixed for weeks now and seemingly every time you want to light it you can't find the matches. You ask the kitchen to locate the nearest box for you: there's one in the cutlery drawer. You've had enough though, so you direct the kitchen to factor the stove repair into your budget. Your stove knows not to hassle you again.

Having enjoyed your meal, you turn on the television but during the first ad break a scrolling message from the kitchen appears at the bottom of the screen telling you that there's an open can of tomatoes that's been getting warm for over two hours. You swear briefly, but are at least glad the house didn't interrupt while you were busy. It knows you're not watching an important show and it did have the decency to wait for an ad break. You go to the kitchen and put the can into the fridge, pausing briefly to put the matches back on the fridge where you expect them.

Three days later you wake up and struggle to the kitchen for a cup of coffee. As you grab the milk, you see the fridge's display panel has a number of messages for you. You'll deal with the emails later but notice that the fridge is complaining that there is a can of tomatoes that is getting beyond its prime. At first you can't find them, but the fridge locates them behind the last of the beer, and you grab the can and blend them. Enjoying your tomato juice with your coffee, you begin a casual cleanup and throw the empty can into the recycling unit.

The recycling unit strips any personal information from the can, and noticing the alloy content ensures it gets picked up for recycling. Some time later the can is shipped to Germany for recycling.

4. Disposable Interaction

Examining this scenario, and the state of hardware technology today, it seems that the production of such processors and network interfaces is practical, if not yet commercially viable. The wide range of devices involved, from the smart can to the local supermarket's CPU cluster, might require a heterogeneous network, with the peripheral processors using different protocols (and physical media) to the Internet backbone. We assume that arbitrary connectivity is feasible, with the possible use of proxies or gateways as required.

Given that this is the case, our current interaction paradigms could, by simple extension, support the proposed scenario. Or could they?

Messaging, RPC and multicast can all be termed *directed* communication models: the destination of the message is specified at the time it is sent (in the case of multicast, this specification is not a single address, but a group or channel upon which the senders and receivers have previously agreed). The problem with requiring knowledge of the destination is that sometimes you don't have it, and this has led to the development of numerous methods of obtaining addresses

- use standardized names, a name server, and a reserved address for local name servers, ie. [GAO90], or
- use LAN segment broadcast or a reserved multicast address to find named objects, ie. [CG85], or
- use a yellow pages service at a reserved address, and select one of the available services in the required class by its advertised properties [OMG97], or
- perform a multicast request to a reserved group, and have all services listen to that group and respond if they can provide the requested function [VGPK97], and work in progress on [GPVD99].

This list is only superficially representative; resolving addresses for directed communication has absorbed a great deal of distributed systems research over the past decade. And yet none of these approaches really solve the problem. Each of them merely shifts the required knowledge to a level of indirection, without addressing the basic issue: that the originator of the message must know where it is to be sent.

In a system where we seriously expect quadrillions of computers, and several orders of magnitude more active endpoints (or objects), **and** where the set of these relevant to an individual is in constant flux at rates of up to hundreds per second, requiring that the sender of a message always specify its destination does not appear feasible.

We propose an alternative that will exist alongside directed communication to ameliorate this problem: *undirected*communication is that where the sender of the messages does not specify their destination.

How can this work? By using a "pull" style, content-based selection of messages. Content-based addressing is not new. It has been widely used in specific applications, and was first popularized (to our knowledge) as a general communication mechanism by Gelernter's Linda [GB82]. It can easily, if inefficiently, emulate directed communication, leading some to propose it as a universal communication model. We prefer to use it in conjunction with directed forms of communication, selecting the model most appropriate for the task at hand.

For content-based addressing to work, message *consumers* (destinations) must have a way to specify that they want to receive a certain class of messages. This information is then used by the infrastructure to route the appropriate messages to the consumer. For the consumer to select a message from a producer (or source), it must somehow describe the message it is to receive. If this description is reduced to its simplest form, it effectively becomes a multicast address: a single, unique attribute used to identify a class of messages.

But using a single, unique attribute to identify messages offers no advantage over directed communication. While ultimately the consumer must share some knowledge with the producer(s), this knowledge can be structured to provide a flexible means of identifying pertinent messages by specifying selection criteria expressed in terms of the message's contents.

In Linda, these specifications are called *templates* and they describe the number, type and order of the message's attributes. The value of a particular attribute can be fixed by providing a value, or is otherwise constrained only to the required data type. Notification services also provide a degree of undirected communication. Unlike Linda, notifications are transient, and without Linda's requirements for persistence, notification services scale to support a much greater overall bandwidth. MIT Athena's Zephyr [DEFJKS88] was followed by PEN [DB92], Rendezvous [OPSS93, TSS95], Keryx [Low97], Elvin [SA97] and others in this general domain.

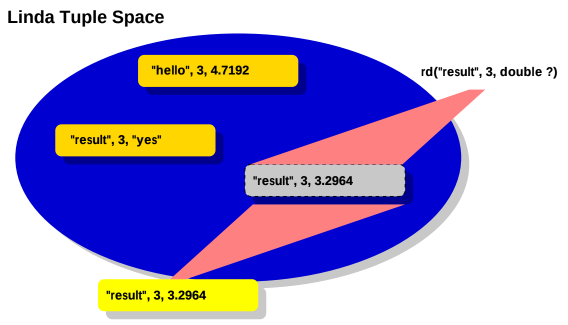

Figure 1: Linda's *rd()* copies a tuple matching the supplied template.

In the terminology of Rosenblum and Wolf [RW97], the directed-ness of notification forms the *naming model*, where classes of events are named using either a structured name, or a property-based name. The degree of direction extends from a multicast address (very directed), through a filter-able structured name, to a property-based query (least directed).

Channel-based services use structured naming. While requiring producers to nominate a specific channel (often a hierarchical name of the form *foo.sub-foo.sub-sub-foo*), they typically allow wildcard filtering of channel names, and often some local secondary filtering of other distinguished attributes.

Keryx and Elvin (described more fully in the following section) use a boolean constraint language to select messages by their content. The messages are *self-describing*, with unordered attributes identified by name, and having a strongly typed values. They allow, for example, selection using numeric ranges and regular expressions on string values. While this mechanism still requires that the message producer and consumer are coupled by the definition of the attribute names, it is significantly more flexible than the other schemes. This has a number of practical benefits for distributed systems.

The deployment of distributed systems is hampered by the close coupling of components through rigid interfaces. Direct, point-to-point binding of components inhibits runtime substitution, removal or addition of components. Using undirected communications, components can be introduced or replaced without affecting any others.

In addition to limiting the interaction architecture of distributed systems to a client-server paradigm, the static definition of component interfaces using an IDL (ONC [Sun88, MS91], DCE [SHMO94], CORBA [OMG91], DCOM [Tha99]) severely restricts the ability of applications to adapt to changes in their environment. An endpoint is bound directly to a component, and cannot be implemented by a group of cooperating objects nor can components simply extend their functionality to include new behavior. Their API effectively dictates the structure of applications.

In a world of disposable computing, where the applications architecture must adapt to the constantly changing environment, interfaces must be able to split and merge, run on a single machine or be spread across the world. Running applications must be able to constantly and seamlessly *adapt* to their current context. And the use of directed communications makes this all but impossible.

The next sections discuss the Elvin architecture and implementation in detail, describing both its current form and the work currently under way to extend it to provide a ubiquitous content-based routing infrastructure for disposable computing.

5. Elvin Architecture

Elvin is a content-based message routing system under development at DSTC. It provides undirected communication, using content-based subscriptions to route self-describing messages.

5.1. Overview

In essence, Elvin routes undirected, dynamically typed messages between producers and consumers. Messages consist of a set of named attributes of simple data types. Consumers subscribe to a class of events using a boolean subscription expression.

Elvin can be described as a pure notification service [RDR98]. Producers push messages to the service, which in turn delivers them asynchronously to consumers. When a message is received at the service from a producer, it is compared to the registered subscription expressions for all consumers and forwarded to those whose expressions it satisfies (see figure 2). Elvin is a dynamic system: messages can be sent without pre-registration of message types and subscriptions can be added, modified, or deleted at whim.

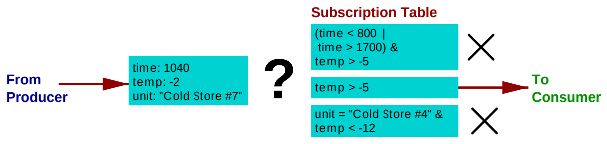

Figure 2: Evaluation of subscription expressions.

The system is implemented as a server daemon that provides the subscription registry and evaluation engine. Client libraries map the wire protocol to programming languages. As well as workstations and personal computers, we are starting to experiment with devices like Palm Pilots and PIC/AVR-class embedded micro-controllers, using radio, IR and wired serial communications to the server.

The flexibility of distributing events based on content is often sacrificed by notification services due to a perceived lack of efficiency [WWWK95]. Common alternatives are to use named channels [DEFJKS88, RBM96, OPSS93, TSS95] or event types [OMG98, Sun99] that must be specified by both producer and consumer. A key benefit of content-based addressing is the reduction of this coupling between producers and consumers. A producer in a channel-based system must be made to send to multiple channels if more than one class of consumer requires the event. Content-based addressing allows any number of different consumers, including those previously unknown, to receive information based on what they need, rather than where the information was directed.

Once producers are freed of the responsibility to direct communications, the determination of the significance of message becomes less important: they can promiscuously send any potentially interesting information, and rely on the system to discard messages of no (current) interest to consumers.

5.2. Quenching

While decoupled message production and consumption is useful, situations where the cost of message generation is significant or the volume of traffic very large, require a "back channel" from the consumers that can be used by producers to determine interest in classes of messages.

The Elvin *quench* facility (named for its ability to reduce message traffic), enables producers to be told when a consumer (or consumers) has subscribed to messages with particular attributes, and optionally obtain the range of values requested. The producer specifies the attribute names that must be present in the subscription expression and the names of attributes for which they want to know the set of requested values. This information is forwarded to the producer whenever changes to the server's subscription base alter the specified values. The quench facility is thus effectively a subscription to messages describing changes to (or initial state of) an Elvin server's subscriptions.

Consider a producer that emits a large number of messages that at any given time might not be of interest to a consumer. By examining the registered subscriptions, it can determine when its information is of interest to a subscriber (or many subscribers) and control its emission.

Alternatively, if it is too expensive to generate unwanted messages, the quench facility can control generation. In the scenario from section 3, consider the supermarket and some packets of chewing gum: the gum is very cheap, so cheap that the manufacturer can only afford to put passive location tracking in the packaging. However, chewing gum is a prime target for shoplifting, so the store wants to track the packets to enable them to detect attempts at theft.

Of course, there are thousands of similar packets in the store, and tracking each of them is well beyond the capacity of their radio location system. Fortunately, only a relatively small number of those packets are removed from the shelves at any one time. What is required is a mechanism enabling the location tracker to determine which packets are of interest.

In figure 3, the theft detector has registered two subscriptions: one for removal of items from the shelves, and another for the sale of items from the cash register (step 1). The radio locator requests quench information for subscriptions to location events (2). After being notified by the shelf that a packet of gum has been removed (3), the theft detector subscribes to notifications of its location including the unique identifier for the packet (4).

The radio locator needs to know what items to track, without directly coupling it to the theft detector (or any other system requiring location information). It needs to examine the active subscriptions to determine for

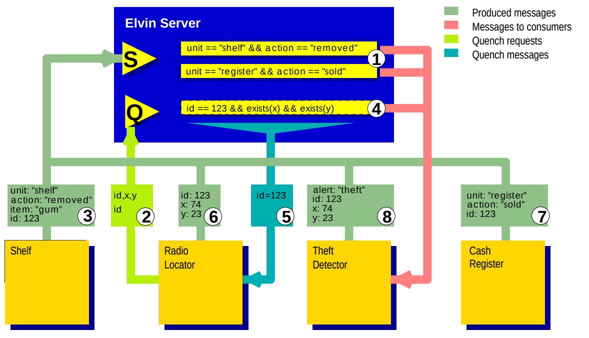

Figure 3: Using *Quench* to control message generation.

which items location events are of interest. The theft detector's subscription (4) matches the quench request from the radio locator (2), and the *id* attribute value is forwarded (5). The radio locator begins tracking the gum, and emitting location messages (6).

Finally, either the gum is sold, and the cash register's sale message (7) informs the theft detector that it need no longer monitor the item, or, if the location coordinates move outside an approved range, the theft detector can emit an alarm (8).

Using the quench facility in this way, producers are able to determine consumers' requirements without losing the flexibility that the decoupling of message production from consumption gives.

6. Elvin 3

Elvin3 is a publicly available implementation of the Elvin architecture, and has been in use for nearly two years. It uses a single TCP/IP-based protocol and provides a simple implementation of quenching. Client libraries are available for C, Java, Python, TCL, Common and Emacs Lisp, and Smalltalk. The initial design criteria targeted the implementation at servicing desktop notification service clients in a LAN environment, from which a scale of around a thousand concurrent clients each with around ten subscriptions was determined. Our chief assumption was that changes in subscriptions would be orders of magnitude less frequent than messages, and the resultant system architecture is heavily biased towards rapid evaluation against a relatively static subscription base.

The client API is simple, consisting, for example, of 11 functions in C. Aside from the initial connection, all server interactions are asynchronous, with notification and subscription quench delivery normally handled via callback functions. Each subscription can also specify multi-threaded delivery, using a pool of threads to run the callback function. A polling API is available, but has been used only for the Smalltalk binding where Elvin's use of native threads did not integrate with the runtime system.

The Elvin3 subscription language is also simple, styled after the C expression syntax, with a handful of predefined functions available for testing attribute existence, (dynamic) type checking, and regular expression matching. Subscription expressions are supplied as strings via the client API and compiled by the server. Multiple subscriptions can be registered using a single connection, and delivered notification messages contain a list of the matching subscriptions whose callback functions are then invoked.

```
conn   ← elvin_connect(how, host, port, quench_cb, err_cb, poll);
         elvin_disconnect(conn);
         elvin_notify(conn, notification);

sub_id ← elvin_add_subscription(conn, sub_expr, nfy_cb, n_thds);
         elvin_replace_subscription(conn, sub_id, sub_expr);
         elvin_del_subscription(conn, sub_id);

         elvin_free_quench(quench);
         elvin_replace_quench_cb(conn, quench_cb);

sockfd ← elvin_get_socket(conn);
         elvin_dispatch(conn);
         elvin_poll_notify(conn, sub_id, notification);
```

Figure 4: Elvin3 C API summary.

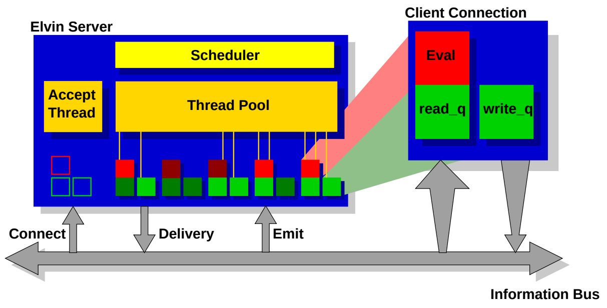

Figure 5: Elvin3 server architecture.

The server architecture is focussed on rapid evaluation and delivery, within the constraints of the service semantics. In keeping with our philosophy of simplicity, the fundamental semantic is "at most once" delivery. After some initial experimentation, and feedback from application programmers, this was extended to guarantee that for a given producer connection, the messages seen by a consumer would retain their order of sending on delivery. Out of order delivery, while enabling lower latency, complicated client programming to an unacceptable degree and the server architecture was modified to support this guarantee.

6.1. Performance

One of the chief goals of the Elvin project has been to demonstrate that content-based routing was performant. Existing work in notification services had uniformly chosen to use a channel-based routing approach, which enabled direct use of IP multicast as a routing optimization.

Operational use of Elvin 3 has satisfied this goal: the flexibility engendered by the simple API and subscription language has led to a wide variety of uses with completely satisfactory performance. But quantitative performance measurement is more difficult.

Simple measurements of end-to-end latency show a wide variance, and don't reflect the possible throughput of the server's threaded evaluation engine. It is here that the most difficulty arises: subscriptions are compiled (with some optimization) when registered with the server. The complexity of the registered subscriptions has direct impact on the delivery latency, as does the CPU load on the server host.

Until a comprehensive benchmarking suite for measuring the performance of content-based routing services is developed, it is most useful to measure server performance in terms of "matches per second" where a "match" is a comparison of a message's attribute value against a subscription's requirement.

An Elvin3 server on an AlphaStation 4/255 workstation can perform approximately 200,000 attribute matches per second, and sustain a throughput of 20,000 messages per second (with 50 active subscriptions and a 10% success rate in subscription evaluation).

6.2. User Experience and Issues

After initial testing within the development group, and then wider exposure within our organization, Elvin3 has been deployed across a wide range of external environments. Aside from the usual bug fixes, very few changes have been required, excluding the addition of the polling interface, and the delivery order guarantee discussed above.

Feedback from application programmers was very positive, with the majority of additional comments suggesting extensions to the basic service. Most common amongst these was a nebulous concept unfailingly called "reliability".

Reliability of undirected communications is a difficult concept to define, let alone implement. But Elvin3's asynchronous API and absence of "feedback" to the producer seems to cause a degree of unease in application programmers.

Various degrees of message reliability are possible: from a simple acknowledgement from the server that it has received the producer's message, through to an assertion that all eligible subscribers registered at the time of delivery have confirmed reception of the message. While either of these is simple enough to implement for a single server, we have two primary concerns: that performance would suffer significantly given the overhead of managing acknowledgements and possible attempts to resend lost messages, and perhaps more importantly, that once extended beyond a single server, implementation of anything other than the initial server's reception of the message seems infeasible.

The second most common request is for security (yet another nebulous concept). This was not unexpected, and our progress on a solution is discussed later.

The quench facility in Elvin3 is primitive: the server simply sends a string containing the or-ed subscription strings of all registered consumers. The Python language mapping comes with a set of classes that encapsulate this string, and provide some higher-level manipulation. These classes have been used by a few applications, and this functionality will be merged into the standard API.

Examining the use of Elvin, there are a significant number of applications classified as "one-shot producers": the end result of the application is the emission of a single message. Obviously, the overhead of establishing a TCP connection (and the resultant resources within the server process) is significant compared to sending a single message. However, TCP's reliability is ubiquitous and it is not clear that an alternative reliable protocol would have substantially lower overhead.

Finally, another significant class of applications is what we call *correlators*: subscribers that wait for a specific combination or sequence of messages, possibly within some time constraints, and produce summary messages when their conditions occur. While these applications consume very few resources, they must be long-lived, and ensuring that all the required processes are restarted with the machine, and remain alive is a significant administrative burden. We are investigating a single daemon process that could have registered descriptions of message sequences and timing constraints, and a specification of how to produce the summary message.

7. Elvin 4: Content-based Routing

Elvin3 is a notification service, designed in the tradition of distributed systems, along with name and yellow pages services and an RPC abstraction. But as our understanding of the undirected communications paradigm grew, we began to make the important semantic distinction between a notification service, and a content-based routing infrastructure. Even before starting Elvin3, we had planned to experiment with federation of notification servers, allowing internet-wide subscription and event notification. We now had multiple sites with local Elvin servers wanting to share traffic and the resulting issues of large numbers of users, administrative ownership of servers and their traffic, server redundancy and the like.

In addition, after using notification for communications between desktop applications, it became increasingly apparent that a wealth of activity outside the desktop computer and local area network was useful if made available as notifications.

Both of these directions were instrumental to our developing view of content-based message routing as a fundamental communications paradigm; a similar abstraction to messaging or RPC and critical to the development of disposable computing.

7.1. Experiments with Federation

A single Elvin3 server can handle at most a few thousand concurrent client connections, and while changing to use a connectionless communications protocol would remove the immediate problem, servicing more than a few thousand clients would approach the limits of the host capacity anyway. The real solution is to extend the service beyond a single server, establishing a federation of autonomous servers cooperating to route messages to their consumers.

How will the properties of a federated service have to differ from those of a single server? A single Elvin server provides *universal availability*: a message from a producer is available for delivery to any subscriber (subject to the security scheme described later). But where the Elvin service crosses an enterprise boundary, some filtering of the traffic might be required in a similar fashion to firewalls used at the IP level. While it should be *possible* to receive traffic from any connected server, not all domains will make all messages available.

A single server also provides *immediate visibility*: a new subscription registered by a client is guaranteed to receive a matching message sent as the next packet on a client's connection. It is not feasible to maintain this semantics on a wide-area scale: it would require synchronization of changes to subscription registries. The delayed propagation of both messages and subscriptions mean that this guarantee cannot be maintained for clients connected to different servers.

Finally, the routing of messages between servers introduces the possibility of messages from a single producer using multiple paths to reach a consumer, and hence arriving at a consumer out of order or duplicated. The Elvin3 server is architected to ensure ordering is maintained, and so explicit measures must be taken to carry this over to the service as a whole.

After some experimentation using client programs to filter and forward traffic between Elvin servers, we have settled on two distinct scenarios for federation. They are distinguished mostly by usage requirements, with different trade-offs taken to address these issues in the two contexts.

7.1.1. Local Area Federation

Within an organization, business unit or site, federation usually requires universal availability, and is used as a means of providing reliability, scaling to large numbers of clients or to provide separate administrative authority over a sub-domain. Automatic failover to backup servers, load-sharing ability and flexible configuration are the dominant requirements. Within a local area latency is significant.

If a produced message is effectively multicast to a cluster of Elvin servers, each of which supports a group of subscribers, supporting large numbers of consumers is simply a matter of balancing the consumer connections evenly across a cluster of servers. This mechanism will scale to an almost indefinite number of consumers. Servers have a hand-over facility, allowing a single, advertised server to balance the client load within the cluster. This facility is also used to perform handover of clients for graceful shutdown.

For ease of administration, connections between servers within a local domain are not subject to topology constraints. To ensure messages are not duplicated, regardless of the inter-server links, each message is tagged with sufficient information to detect duplicates which are then discarded. Links between servers are unidirectional, and have optional filters to control message propagation.

7.1.2. Wide Area Federation

Beyond the bounds of an enterprise domain, access to messages is the primary requirement; a communications "backbone" allowing subscription to messages sent from anywhere in the world (or campus, or company) and publication of internal messages for global access.

The primary concern in routing messages beyond a local domain is scalability. Both the traffic volume and the computational effort required to route it must scale to support our quadrillion nodes, many of which will host multiple Elvin clients.

Obviously, simply forwarding all traffic from a local domain onto a global message bus is infeasible. Ideally, only those messages that exactly match the requirements of one or more subscribers, somewhere on the global network, should be sent on. In effect, the backbone should subscribe to a set of messages from a local domain.

However, it should also be possible to prevent both the export and import of classes of messages. An administrator of a domain must be able to apply a filter at the domain boundary, protecting private information from dissemination and restricting the visibility of external events within the domain.

Design of the backbone protocols is still an area of active work. In particular, issues of mobile users and the equivalent of the "Slashdot Effect" [Adl99], where millions of consumers want access to a single message stream, present extreme challenges to the routing infrastructure.

A content-based service has one advantage in scalability; total load on the system is shared across the federation. Each node of the federation only deals with the data once, unlike point-to-point protocols where the originating endpoint must process every request.

7.2. Elvin 4

Elvin4 is an evolution of the Elvin architecture, with refinements across the board from protocol to API. Major changes include

- the introduction of a security mechanism,
- a modular architecture for the underlying marshalling, security and transport protocols,
- automatic server location using SLPv2 [GPVD99],
- better quenching support, and
- an extended subscription language, including support for international strings.

With the facility for multiple protocol stacks supporting the high-level communication, comes the requirement for an interworking protocol to ensure that all Elvin domains can interconnect if required. The combination of XDR [Sri95] marshalling, SSL-3 [FKK96] security and TCP/IP transport has been defined as the standard protocol stack, which must be used for to connect to the Elvin backbone.

Despite a more complex internal architecture, we expect significant performance gains from this latest implementation. Careful memory management, a revised threading strategy and better marshalling are targeted at improving the server’s bandwidth. Additionally, merging and sharing evaluation graphs [GS94] may lead to signifcant performance increases.

7.2.1. Security

The basic requirements for securing undirected communications are simple: firstly to prevent unauthorized subscribers from receiving messages, and secondly to prevent attackers from "spoofing" messages from a legitimate producer.

A third requirement is introduced by the Elvin quench mechanism. The returned quench messages must not reveal subscriptions for which the producer may not produce matching messages.

In order to retain the loose coupling of content-based routing, we have adopted a mechanism derived from [Pin92], attaching keys transformed by a one-way function to each sent message. Producers retain the raw key, and distribute the transformed key to authorized consumers. When sending a message, the raw, private key is presented, and transformed by the server on arrival. Consumers supply their (already transformed) key when subscribing, and the server compares keys as part subscription evaluation.

Privacy of both the authorization keys and message content can be preserved by encryption of the link between the client library and the server (and between federated servers). Users can specify their preference for security mechanism, authentication and privacy during connection establishment. The use of authentication and privacy is optional, and computationally expensive.

The major problem with this mechanism is that the plaintext of the messages, is exposed to the intermediate servers routing the message to its destinations. Unfortunately, when using the content to perform the routing, this is unavoidable. Of course, it is always possible to encrypt the body of the message prior to transmission if required.

7.2.2. Charging

While we anticipate that most use of Elvin services will be "local" and remain uncharged, the provision of information services and federation of Elvin domains requires a charging model allowing producers to add a premium to the basic transportation costs and backbone routers to allocate forwarding costs to users.

To complicate matters, the cost of routing is not consistent, with complex subscriptions are able to consume significant CPU resources during evaluation. While a simple model of charging by number of bytes is attractive, it does not allow for accurate cost recovery. Additionally, it is not clear that a single charging model will suffice: allocation of the total cost between the producer and consumers of a message could occur in any number of ways, with neither producer only nor consumers only acceptable.

Note that billing is not part of the problem. The Elvin server must simply log the data required for billing which can be processed by a third party.

A simple charging mechanism is provided in Elvin4, but charging in a wide-area Elvin federation remains an open issue.

8. Future Work

Elvin4 is a testbed for our research into the challenges of internet-scale undirected communication using content-based routing. Active work proceeds on federation protocols, the security and charging mechanisms and additional services.

An undirected communication infrastructure would be incomplete without some form of correlation engine (see [LV95]) providing recognition of message patterns. Leveraging previous work in Linda on such recognition, complex correlations can be built from smaller components to embed expert knowledge into the network, for example using a process trellis [FG91].

The availability of access to such a wealth of information from so many devices makes management of an object's relevant context an extremely difficult problem. While the use content-based routing and a correlation service make it relatively simple to create the mechanism for contextualization, the real problem lies in creating policy of sufficient detail for everyday use. People are extraordinarily good at casual awareness and selective focus. The challenge is firstly to simplify mechanisms for defining detailed policy, and secondly to ensure the portability of that policy so it adapts as location changes. We are drawing from related work on awareness and context management in computer supported cooperative work [MKFPFT97, Fit98] to provide objects with the ability to adapt to their surroundings in similar ways to people.

9. Conclusion

Content-based routing is a fundamentally different paradigm for interaction between networked objects. By removing the necessity for producers to direct messages, we gain enormous flexibility in system architecture and scalability over traditional communication systems allowing us to provide an interaction environment for disposable computing.

We are only at the beginning of our exploration of this paradigm but can already see the benefits of decoupling the production, consumption, and dissemination of data between networked components. Undirected communication facilitates systems that are more easily extended, simpler to componentize, and contain a clearer mapping to real world interactions between objects.

Disposable computing requires a revolution in distributed systems; existing paradigms will not scale effectively to support the rapidly changing environment of vast numbers of relevant objects. Undirected messaging overcomes some of the problems of existing systems and provides a viable infrastructure for communication in a quadrillion node network.

And besides, how else will we know where our tomatoes are?

Availability

Elvin is available in both source and binary form under a not-for-commercial-use license. Full documentation, FAQs, additional software and the download itself can be found on the Elvin homepage

http://www.dstc.edu.au/Elvin/

The SLPv2 implementation used in Elvin will also be available independently. See

http://www.dstc.edu.au/Elvin/Sulphur/

Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centre Program through the Department of Industry, Science and Resources of the Commonwealth Government of Australia.

This work has also been supported in part by the United States Defense Advanced Research Projects Agency under grants F30602-96-2-0264 and F30603-94-C-0161 (both administered by the US Air Force through Rome Laboratories). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

Adl99.
Stephen Adler, *The Slashdot Effect: An Analysis of Three Internet Publications*, http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html 1999.

CG85.
Bill Croft and John Gilmore, "Bootstrap Protocol (BOOTP)," IETF Request for Comments, RFC-951 (September 1985).

DB92.
DiBella and Bhandaru, "Pilgrim Event Notifier Version 1.0," Technical Report, University of Massachusetts, Amherst, MA (November 1992).

DEFJKS88.
DellaFera, Eichin, French, Jedlinsky, Kohl, and Sommerfeld, "The Zephyr Notification Service," *Proceedings USENIX Winter 1988,* pp. 213-219 (1988).

FG91.
Michael Factor and David Gelernter, "Software Backplanes, Realtime Data Fusion and the Process Trellis," YALEU/DCS/TR-852, Yale University Department of Computer Science (March 1991).

Fit98.
Geraldine Fitzpatrick, *The Locales Framework: understanding and Designing for Cooperative Work,* PhD Thesis, University of Queensland, Australia (1998).

FKK96.
A. Frier, P. Karlton, and P. Kocher, *The SSL 3.0 Protocol,* Netscape Communications Corporation (Nov 18, 1996).

GAO90.
S Gursharan, R Andrews, and A Oppenheimer, *Inside AppleTalk,* Addison-Wesley (1990). Second Edition

GB82.
Gelernter and Bernstein, "Distributed communication via global buffer," *ACM Symposium on Principles of Distributed Computing,* pp. 10-18 (August 1992).

GPVD99.
Erik Guttman, Charles Perkins, John Veizades, and Michael Day, "Service Location Protocol, Version 2," IETF Internet Draft, work-in-progress, draft-ietf-srvloc-protocol-v2-12 (February 1999).

GS94.
John Gough and Glenn Smith, "Efficient recognition of events in a distributed system," *Proceedings 18th Australian Computer Science Conference,* (1994).

Low97.
Colin Low, "Integrating Communication Services," *IEEE Communications* **35**(June 1997).

LV95.
David C. Luckham and James Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering* **21**(9) pp. 717-734 (September 1995).

MKFPFT97.
Tim Mansfield, Simon Kaplan, Geraldine Fitzpatrick, Ted Phelps, Mark Fitzpatrick, and Richard Taylor, "Evolving Orbit: a progress report on building locales," *Proceedings of Group97,* ACM Press, (November 1997).

MS91.
Chuck McManis and Vipin Samar, *Solaris ONC: Design and Implementation of Transport-Independent RPC,* Sun Microsystems, Inc (1991).

OMG91.
Object Management Group, "Common Object Request Broker: Architecture and Specification," OMG TC Document 91-12-1 (December 1991).

OMG97.
Object Management Group, "Trader Object Services Specification," OMG TC Document 97-12-23 (December 1997).

OMG98.
Object Management Group, "Notification Service: Joint Revised Submission," OMG TC Document telecom/98-11-01 (November 1998).

OPSS93.
Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen, "The Information Bus: an architecture for extensible distributed systems," *ACM SIGOPS Operating Systems Review* **27**(5) pp. 58-68 (December 1993).

Pin92.
James Pinakis, "Directed Communication in Linda," *Proceedings 15th Australian Computer Science Conference*, pp. 731-743 (January 1992).

RBM96.
Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis, "Horus, a flexible Group Communication System," *Communications of the ACM*, (April 1996).

RDR98.
Ramduny, Dix, and Tom Rodden, "Exploring the Design Space for Notification Servers," *Proceedings Computer Supported Cooperative Work (CSCW'98)*, pp. 227-235 ACM, ().

RW97.
David S Rosenblum and Alexander L Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," *Proceedings of the Sixth European Software Engineering Conference/ACM SIGSOFT, Fifth Symposium on the Foundations of Software Engineering*, (September 1997).

SA97.
Bill Segall and David Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proceedings AUUG Technical Conference (AUUG'97)*, pp. 243-255 (September 1997).

SHMO94.
John Shirley, Wei Hu, David Magid, and Andy Oram, *Guide to Writing DCE Applications*, O'Reilly and Associates (May 1994). 2nd Edition

Sri95.
R. Srinivasan, "XDR: External Data Representation Standard," IETF Request for Comments, RFC-1832 (August 1995).

Ste92.
Neal Stephenson, *Snow Crash*, Bantam (1992).

Sun88.
Sun Microsystems, Inc, "RPC: Remote Procedure Call Protocol Specification, Version 2," IETF Request for Comments, RFC-1057 (June 1988).

Sun99.
Sun Microsystems, Inc, "Jini Distributed Event Specification," Technical Report (January 1999.).

Tha99.
Thuan L Thai, *Learning DCOM*, O'Reilly and Associates (April 1999). 1st Edition

TSS95.
Teknekron Software Systems, *Rendezvous Software Bus Programmer's Guide*. 1995.

VGPK97.
John Veizades, Erik Guttmann, Charles Perkins, and S Kaplan, "Service Location Protocol," IETF Request for Comments, RFC-2165 (June 1997).

Wei91.
Mark Weiser, "The Computer for the Twenty-First Century," *Scientific American*, (September 1991).

WWWK95.
Jim Waldo, Wollrath, Wyant, and Kendall, "Events in an RPC Based Distributed System," SunLabs Technical Report SMLI TR-95-47 (November 1995).