Voluntary Cooperation in Pervasive Computing Services

Mark Burgess and Kyrre Begnum – Oslo University College

ABSTRACT

The advent of pervasive computing is moving us towards a new paradigm for computing in terms of *ad hoc* services. This carries with it a certain risk, from a security and management viewpoint. Users become increasingly responsible for their own hosts. A form of service transaction based on minimal trust is discussed. A proof of concept implementation of non-demand (voluntary) services is discussed for pervasive computing environments. 'Voluntary Remote Procedure Call' is a test-implementation of the proposed protocol integrated into cfengine, to show how voluntary cooperation of nodes can allow a cautious exchange of collaborative services, based on minimal trust. An analysis of implementation approaches followed by a discussion of the desirability of this technology is presented.

Introduction

Pervasive or ubiquitous computing is often wedded to the future vision of mobile and embedded devices, in smart homes and workplaces; however, pervasive computing environments already exist today in Web Hotels and at Internet Service Providers. A changing base of customers meets in an environment of close proximity and precarious trust and offers services.

Pervasive mobile computing presents two independent challenges to content services. The first concerns how to secure platforms for virtual commerce. Here the problem is that one does not possess guaranteeable credentials for those connecting to the services. The second concerns how to deal safely with anonymous, non-commercial services offered entirely on an ad hoc, cooperative basis. Here the problem is the risk involved in interacting at all with a client of unknown intentions.

The former generally drives discussions about security and public key infrastructures (though this is of little help unless every user is independently verified and identifies cannot be forged). The latter has made a name for itself through peer to peer file sharing such as Napster, Gnutella and other services. There is currently no technological solution that can determine the intentions of a client attempting to connect to a service.

The challenges of pervasion are not only technical, but also 'political'. *Autonomy* is a noose by which to hang everything from security to system management in coming years. In the foreseeable future, one can expect pervasive services to be vying for attention on every street corner and in every building. The key ingredient that makes such a scenario difficult is *trust*. Orderly and predictable behaviour is a precondition for technology that performs a service function, but if one connects together humans or devices freely in environments and communities that share common resources, there will always be conflicting interests. Anarchy of individual interests leads to contention and conflict amongst the members of the community; thus, as in the real world, an orderly policy of *voluntary cooperation* with norms is required that leads to sufficient concensus of cooperative behaviour.

In this paper, we shall be especially interested in the administrative challenges of pervasion, such as services like backup, software updates, policy updates, directory services etc. We shall attempt to tackle a small but nevertheless important problem, namely how to 'approach' peers for whom one does not have automatic trust.

In a pervasive setting, the the sheer multiplicity of devices and hosts demands autonomous administration, for scalability. This means that decisions about trust cannot be reasonably processed by a human. Self-managing 'smart' devices will have to cope with decision-making challenges normally processed by humans, about the trustworthiness and reliability of unknown clients and servers. Devices must make political decisions, based on often fuzzy guidelines. This is not a problem that can be solved by public key infrastructures, since apparent certainty of identity is no guarantee for trust.

Virtualization and the Multiplicity of Autonomous Agents

In a few years, most services could well be provided by virtual machines running on consolidated computing power [1, 2], leading to an even denser packing of competing clients and services. There are distinct advantages to such a scenario, both in terms of resource sharing and management. Users can then create and destroy virtual machines at will, in order to provide or consume services or manage their activities.

The ability to spawn virtual clients, services and identities, quite autonomously, makes the interactions between entities in a pervasive environment full of uncertainties. Rather than having a more or less fixed and predictable interface to the outside world, as we are used to today, users will become chameleons, changing to adapt to, or to defy, their neighbours' wishes. What users choose to do with this freedom depends more on the attitudes of the individuals in the virtual society than on the technologies they use.

Today, at the level of desktop computing, the trend from manufacturers is to give each user increasing autonomy of administrative control over their own workstations, with the option to install and configure software at whim. Thus each user can be either friend or foe to the community of behaviours – and the uniformity of policy and configurations that many organizations strive for today will not be a certain scenario of the future.

We are thus descending into an age of increasing autonomy and therefore increasing uncertainty about who stands for what in the virtual society. The aim of this paper is to offer a simple mechanism for host cooperation, without opening the hosts to potential abuses. It does this by allowing each individual agent to retain full control of its own resources.

The plan for the paper is as follows: we begin with a discussion of the need for cautious service provision in a pervasive environment, by using what is known about client-server relationships between humans. Some examples of management services are discussed that present a risk to both parties and it is suggested how a consentual, voluntary protocol might be implemented to reduce the risk to an acceptable level. A solution to negotiating voluntary cooperation is presented, which trades the traditional model of vulnerable guaranteed service levels for increased security at the expense of less predictable service rates. An implementation of the protocol in the programming language Maude helps to enlighten the possible administrative obstacles which follows of the protocol. An example implementation in the system administration agent framework cfengine is presented which incorporates voluntarism in the existing cfengine communication framework. A comparison and discussion of the two approaches concludes the paper.

Client-server Model

In the traditional model of network services, a client stub contacts a server on a possibly remote machine and requests a response. Server access controls determine whether the request will be honoured and server host load increases as a function of incoming requests. Clients wait synchronously for a reply. This is sometimes called a Remote Procedure Call (RPC).

This model is ideal for efficiently expediting services that are low risk and take only a short time to complete, but it has a number of weaknesses for services that require significant processing or have basic administrative consequences for the network: there is thus the risk of exploitation and denial of service vulnerabilities.

Management services are not typical of network services in general. By definition they are not supposed to be high volume, time critical operations such as web services or transaction processing services. They are nonetheless often of great importance to system security. In a pervasive computing environment, hosts are of many different types; many are mobile, partially connected systems under individual control, configured at the whim of their owner. A more defensive posture is therefore called for in the face of such uncertainty.

This paper presents a simple protocol that assumes no automatic trust: a remote service model based entirely on 'pull' semantics, rather than 'push' semantics. Thus services can be carried out, entirely at the convenience of the answering parties: the risk, at each stage, is shifted to the party that is requesting communication. One interesting side-effect of this is that services become more resilient to Denial of Service attacks.

Pervasion and Stable Policies

The interactions between hosts can naturally lead to changes in their behaviour, e.g., if they exchange configurations, policy instructions or even software. In the ad hoc encounters that emerge between politically autonomous hosts, some stability must condense out of the disorder if one is to maintain something analogous to law and order in the virtual realm [3]. Law and order can only be built on cooperative concensus – it can only be enforced if a majority wants it to be enforced, and conforms in a scheme to enforce it. How such a concensus emerges is beyond the scope of the present paper, but will be discussed elsewhere [4, 5, 6].

Tourism

In the present scenario, the model for contact between computers is like virtual tourism: visitors roam through an environment of changing rules and policies as guests, some of them offering and requesting services, like street sellers or travelling salesmen (no relation to the famed algorithmical optimizer). The communication in such encounters is de-centralized; it takes the form of a "peer-to-peer" relationship between participants. The snag with peer-to-peer transactions that it requires considerable trust between the participating hosts. Who is to declare that an arbitrary participant in the scheme is trustworthy [7]?

It is known from game theoretical and evolutionary simulations that stability in groups of collaborative agents is a precarious phenomenon, and thus caution in accepting arbitrary policy instructions is called for. If hosts alter their behaviour automatically in response to ad hoc interactions, then the effective behaviour of the group can easily be subverted by a rogue policy [8] and the illusion of organizational control can evaporate.

The approach to collaboration which is adopted below is to assume the scenario of lawlessness mentioned above, by placing the provision of administrative services (e.g., backups, information or directory services, policy distribution etc.) on an entirely voluntary basis. This is, in a tangential way, a continuation of the idea of the "pull contra push" debate in system administration [9, 10] (see Figure 1).

Voluntary Cooperation

There are three reasons why the traditional model of client-server communication is risky for pervasive computing services (see Figure 1).

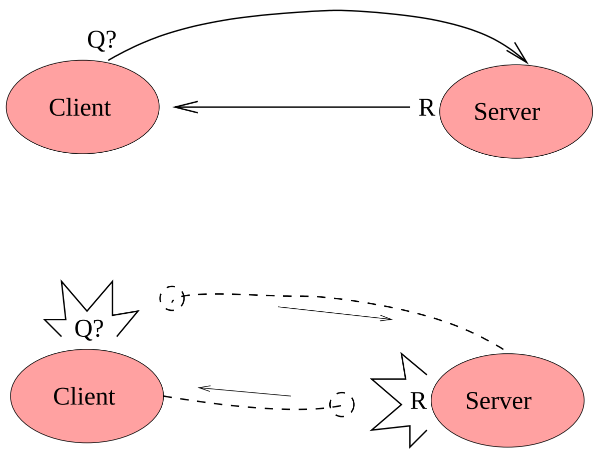

Figure 1: In a traditional client-server model, the client drives the communication transaction "pushing its request". In voluntary RPC, each party controls its own resources, using only 'pull' methods.

1. Contemporary network servers are assumed to be 'available'. Clients and servers do not expect to wait significantly for processing of their requests. This *impatient expectation* leads to pressure on the provider to live up to the expectations of its clients. Such expectation is often formalized for long term relationships as Service Level Agreements (SLA). For ad hoc encounters, this can be bad both for the provider and for the client. Both parties wait synchronously for a service dialogue to complete, during which their resources can easily be tied up and exploited maliciously (so-called Denial of Service attacks).

   Non-malicious, random processes can also be risky. It is known from studies of Internet traffic that random arrival processes are often long tailed distributions, i.e., include huge fluctuations that can be problematical for a server [11]. Providers must therefore over-dimension service capacity to cope with seldom events, or accept being choked part of the time. The alternative is to allow asynchronous processing, by a schedule that best serves the individuals in the transaction.
2. The traditional service provider does not have any control over the demand on its processing resources. The client-server architecture drives the servers resources at the behest of the client. This opens us to risk of direct attacks like Denial of Service attacks and load storms from multiple hosts. In other words, ad hoc encounters do not have to trust only individuals but also the entire swarm of clients in a milieu collectively. The alternative is that hosts should be allowed to set their own limits.
3. The traditional service provider receives requests that are accepted trustingly from the network. These might contain attacks such as buffer overflows, or data content attacks like viruses. The alternative is for the server to agree only to check what the client is offering and download it if and when it can be handled safely.

Although one often aims to provide on-demand, high-volume services where possible, it is probably not prudent to offer any service to just any client on demand, given how little one actually can know about them. Nor is it necessary for many tasks of a general administrative nature, where time is not of the essence, and a reply within minutes rather than seconds will do (e.g., backups, policy updates, software updates etc.). Avoiding messages from a client is a useful precaution: some clients might be infected with worms or have parasitic behaviour.

The dilemma for a server is clearly that it must expose itself to risk in order to provide a service to arbitrary, short-term customers. In a long term relationship, mutual trust is built on the threat of future reprisals, but in an opportunistic or transitory environment, no such bond exists. Clients could expect to 'get away' with abusing a service knowing that they will be leaving the virtual country soon.

In the voluntary cooperation model this risk is significantly reduced in the opening phase of establishing relations: neither "client" nor "server" demand any resources of each other, nor can they force information on each other. All cooperation is entirely at the option of the collaborator, rather than at the behest of the client. Let us provide some examples of how voluntary collaboration might be used,

Configuration Management in a Pervasive Setting

A natural application for voluntary cooperation is the configuration or policy management of networked hosts. How shall roaming devices adapt to what is local policy [12]? A host that roves from one policy region to another might have to adapt its behaviour to gain acceptance in a scheme of local rules, just as a traveller must adapt to the customs of a local country. However, this does not imply automatic trust (see Figure 2). Today, many users think in terms of creating a Virtual Private Network (VPN) connection back to their home base, but to get so far, they must first be allowed to interact with their local environment.

Configuration management is a large topic that concerns the configuration of resources on networked hosts. See, for instance, [14] for a review. There are several studies examining how computing systems can be managed using mobile agents. However, mobile agents have not won wide acceptance amongst developers [15] and they violate the risk criteria of the present paper. Opening one's system to a mobile agent is a potentially dangerous matter, unless the agent follows a methodology like the one described here.

Three examples of autonomy in computer administration are shown in Figure 2. In (a) all instructions for configuration and behaviour originate from a central authority (a controller). This is like the view of management in the SNMP model [16, 17] and traditional telecom management models like TMN [18]. The dotted lines indicate that nodes in the network could transmit policy to one another to mitigate the central bottleneck; however, control rests with the central controller, in the final instance. In (b) there is a hierarchical arrangement of control [19, 20]. A central controller controls some clients individually, and offers a number of lesser authorities its advice, but these have the power to override the central controller. This allows the delegation of control to regional authorities; it amounts to a devolution of power. In (c) one has a completely decentralized (peer to peer) arrangement, in which there is no centre [21, 3]. Nodes can form cooperative coalitions with other nodes if they wish it, but no one is under any compulsion to accept the advice of any other.

This sequence of pictures roughly traces the evolution of ideas and technologies for management. As technology for communication improves, management paradigms become naturally more decentralized. This is probably more than mere coincidence: humans fall naturally into groups of predictable sizes, according to anthropologists [22]. There is a natural rebellion against centralization once the necessary communication freedoms are in place. Then groupings are essentially ''buddy lists'', not central authorities.

A decentralized scheme has advantages and disadvantages. It allows greater freedom, but less predictability of environment. Also, by not subscribing to a collaborative authority, one becomes responsible for one's own safety and well-being. This could be more responsibility than some ''libertarian'' nodes bargain on. One could end up re-inventing well-known social structures in human-computer management.

The immunity model of system administration describes how nodes can made to take responsibility for their own state, in order to avoid flawed external policies, bottle-necked resources and unnecessary assumptions about trust [10]. The trouble with making hosts completely independent is that they need to communicate and even 'out-source' tasks to one another. The immunity model must therefore extend to communications between hosts that allow them to maintain their autonomy, accepting and rejecting data as they please. Other suggestions for an RPC paradigm in a mobile environment include refs. [23, 24, 25].

Some examples applications for voluntary Remote Procedure Call (RPC) are presented below. Most of these address the impossibility of establishing true nature of a client in an ad hoc meeting.

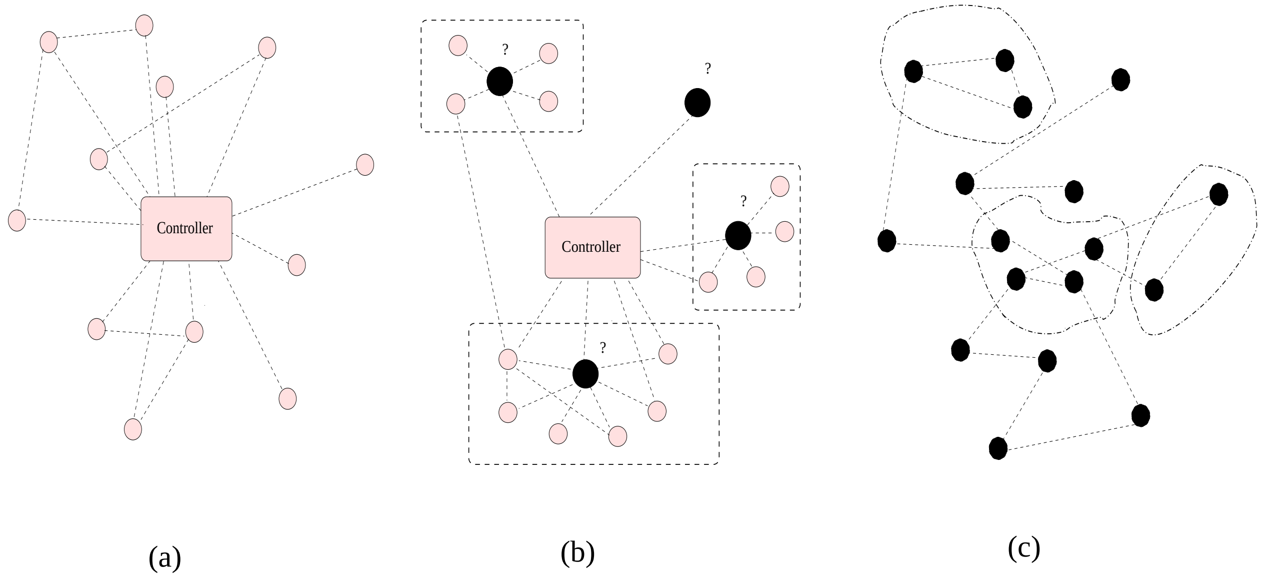

Figure 2: Three forms of management between a number of networked host nodes. The lines show a logical network of communication between the nodes. Dark nodes can determine their own behaviour, i.e., they have political freedom, while light nodes are controlled. The models display a progression from centralized control to complete de-centralization of policy (see [13, 3]).

1. Mobile hosts can use voluntary collaboration to check in to home base for services. For example, a laptop computer that requires a regular backup of its disk to home base, using a pull-only mechanism will need to check in to home base because the server does not know its new IP address, or whether it is even up and running. (Using IPv6, a host could register its mobile address with home base to set up IPv6 forwarding, but if a device has no home base, then this will not work.) The laptop would like to control when its resources are going to be used by the backup software and vice versa, rather than being suddenly subjected to a highly resource intensive procedure at an arbitrary time.

   With a voluntary agreement, the resulting backup time is a compromise between the clients wishes and the server's wishes. It is not exactly predictable in time for either party, but it is predictable in its security. Hosts must learn the expected services times rather than assume them.

   (Again, it is emphasized that encryption cannot solve this problem. Merely opening an encrypted tunnel back to home base does not help the parties to manage their resources. It only helps them to establish their respective identities and to communicate privately.)
2. Exchanges of security alerts and patches could be offered on a voluntary basis, e.g., warnings of recent port-scanning activity, detected anomalies or denial of service (including spam) attacks. Such warning could include IP addresses so that hosts could add the offending IP-port combinations to its firewall. Using this scheme, client hosts agree to signal one another about changes and patches which they have received. These could be cross referenced against trusted signature sources downloaded separately – allowing an impromptu Trusted Third Party collaboration. This kind of technique is now being used for online operating system updates.
3. The ultimate autonomy in distributed services is the free exchange of information and services between arbitrary hosts [12]. Voluntary acceptance of policy or method data could be utilized as a 'cautious flooding' mechanism, for 'viral spreading' of data; e.g., in which each host signals its neighbours about current policy or software updates it has received, allowing them a controlled window of opportunity to download the software (rather than simply opening its ports to all hosts at the same time). It is known that such mechanisms are often efficient at spreading data [26, 27]
4. Neighbouring Service Providers with a standing arrangement with one another could use the voluntary mechanism to sell each other server capacity, using virtual machine technologies. When the demand on ISP1 becomes too high, a method SpawnVM() which is sent to the collaborator ISP2. If ISP2 has the capacity to help, it spawns a virtual machine and returns the IP address. The method call could also include a specification of the system.

These examples underline the need for individuals to maintain control of the their own resources, but at the same time allow one another the potential to collaborate.

Protocol

There are two phases of voluntary cooperation: agreeing to work together (forming a contract) and fulfilling one's obligations (if any) with minimal risk to oneself. Then, given an autonomous framework the implementation of a cautious service protocol is straightforward.

The Negotiation Phase

A willingness to cooperate between individuals must be established in any service before service provision can be considered. This is one of the crucial phases and it brings us back to the opening remarks. Apart from the traditional client request, there are two ways to set up such an agreement: with or without money. One has the following options:

1. A contract based on money: a client is willing to pay for a service and some form of negotiation and payment are processed, based on a legal framework of rules and regulations. This is a common model because it is rooted in existing law and can threaten reprisals against clients who abuse privileges by demanding a knowledge of their identity in the real world.
2. No contract but registration: clients use the service freely provided they register. Again, they must surrender some personal details about themselves. This is a declaration of trust. A password or secret key can be used to cache the results once a real world identity has been established. Public key technologies are well acquainted with the problem of linking true identity to a secret key. We shall not discuss this further but assume the problem to be soluble.
3. Open connection: the traditional way of connecting to host services requires no registration or credentials other than an IP address that may or may not be used to limit access.

In volume services, one usually wants to know in advance which clients to expect and what their demands on service are: this expectation is guaranteed by access control rules, firewall solutions and service quality configuration. In voluntary cooperation models, service agreements cannot be made in the same way, since both parties have responsibilities to one another.

We do not consider the details of the negotiation process here, since it is potentially quite complicated; rather we shall assume, for this work, that a negotiation has taken place. The result of this process is a list of logical neighbours who are considered to be 'method peers', i.e., peers who will execute remotely requested methods.

Service Provision Phase

After intentions have been established, one needs a way of safely exchanging data between contracted peers in such a way that neither party can subsequently violate the voluntary trust model. To implement this, we demand that a client cannot directly solicit a response from a server. Rather, it must advertise to trusted parties that it has a request and wait to see if a server will accept it of its own free choice.

1. Host A: Advertises a service to be carried out, by placing it in a public place (e.g., virtual bulletin board) or by broadcasting it.
2. Host B: Scout A looks to see if host A has any jobs to do, and accepts job. Host B advertises the result when completed by putting the result in a public place.
3. Host A: looks to see if host B has replied.

Seen from the perspective of host A and B, this is done entirely using 'pull' methods, i.e., each host collects the data that is meant for it; no host can send information directly to another, thereby placing demands on the other's resources. The algorithm is a form of batch processing by posting to local 'bulletin boards' or 'blackboards'. There is still risk involved in the interaction, but each host has finer control over its own level of security. Several levels of access control are applied:

- First the remote client must have permission to signal the other of its presence
- Next the server must approves the request for service and connect to the client.
- The client must then agree to accept the connection from the server (assuming that it is not a gratuitous request) and check whether it has solicited the request.

The basic functionalities of this approach may be recognized as typical broker or marketplace approaches for agents in a multi agent system. This makes the discussion particularly interesting because it addresses a relatively known scenario. Before we go into into the administrative challenges of this approach, we present one concrete implementation as a protocol model simulation.

Protocol Specification in Maude

The programming language Maude [28] is specializes in the simulation of distributed systems. In contrast to typical programming approaches, this language is used to state the allowed transitions of a configurations state to another. This opens for searches in the execution domain of the entire system. With its roots in transitional logic it has been used to analyze protocol related to mobile systems and security. The Maude framework enables the writer to focus on the essence in the protocol without wasting to much code on unrelated features. The process is especially helpful if one wants to examine prototype implementations of a protocol, for testing. A secondary effect is that formal specifications of a system often reveal hidden assumptions in the design of the protocol, which lead to differences in the protocol implementation and bugs.

Programming in Maude consists mainly of two tasks: (1) defining the data model in terms of classes and types of messages between objects and (2) defining the allowed transitions from one program state to another. It is thus declarative. Thus, instead of programming a particular path of execution, one sets the scene for all possible steps that might occur without assuming what happened before. Execution of a Maude specification is like rewriting logic.

If one has a starting configuration and a number of independent rewriting rules, one could, in principle, use any of the possible rules. Maude creates a tree, where the initial configuration is the root node and all possible outcomes form the rest. For complex systems it is clear that the human mind can imagine only few of all the possible configurations. Maude possesses a search mechanism to actually traverse this tree to look for certain configurations.

Maude takes an object-oriented approach when designing distributed systems. One usually defines 'class objects' or data-types for the objects that are to communicate with each others, and a class for messages. Below is an example of the classes specified in this implementation: A client (the node requesting service), a server, and a blackboard.

```
class Client | pendingRPC : RPCList,
 knownServers : OidList,
 jobs : JobList,
 completedJobs : JobList,
 rpcEnabled : OidList,
 blackBoard : Oid .

class Server | rpcClients : OidList,
 blackBoard : Oid .

class BB | rpcRequests : RPCList,
 rpcReplies : RPCList .
```

To illustrate the specification of a transitional rule, consider the following:

```
crl [accept-service-request] :
 < S : Server | rpcClients : OL >
 (msg RPCrequestService(C,S) from C to S )
 =>
 < S : Server | rpcClients : OL :: C >
 (msg RPCrequestReply(C,S,true) from S to C )
 if ( not containsElement(OL, C) ) .
```

The "=>" marks the transition from the state before to the state after. The state before says only that there is a Server object *S* and a message addressed to that object. The server object (enclosed in "< >") has a list of object pointers called rpcClients. The client is added to that list if it is going to be served by the server. Note that Maude assumes that there may be many other objects currently in the same state. The only thing that is specified is that, if at a given point a server *S* sees the message, it may react to it. Nothing is stated about time constraints or priorities of transitions.

The displayed rule has a conditional that states that if a server sees a message addressed for it and containing a RPCrequestService(C,S) payload from client *C* it will answer with a reply but only if the client is not served by the server already. In this particular case it will answer "yes" to the request. A similar rule exists where the server answers "no". Maude chooses one of these rules randomly in its simulation.

Execution of a Maude program means supplying an initial state (also called Configuration) and letting Maude choose a path at random, based on the all the possible transitions of the given state. There is also the possibility of issuing searches in the execution tree (all possible paths) of the initial state in order to seek out undesirable states of the protocol. This is beneficial for system designers who work with distributed systems too complex for unaided humans to think of, in every possible outcome. Other fields of computer science have used formal modelling of this kind for a long time. We perceive this tool to be of value in the field of theoretical system administration as well.

Every execution tree is based on an initial configuration. To begin with something basic, lets take the case of one of client and one server. We define simpleInit to be a function that returns the following configuration:

```
*** A simple initial state
eq simpleInit =
< "client" : Client | completedJobs : empty,
                    pendingRPC : nil,
 blackBoard : "BB",
 rpcEnabled : none,
         knownServers : "server",
         jobs : "job1" ^ "job2" >

< "server" : Server | blackBoard : "BB",
 rpcClients : none >
< "BB" : BB | rpcReplies : nil,
 rpcRequests : nil > .
```

All participants know about each other out-of-band, and so they start with variables pointing to each other. The client has two jobs it needs to be performed, "job1" and "job2". It has an empty list called completedJobs and the most basic search is to query whether this list can contain both jobs in the end. That would be a sign that the specification works as intended:

```
(search [1] simpleInit =>+
 < "client" : Client |
 completedJobs : "job1" ^ "job2",
 Client:AttributeSet >
 C':Configuration .
)
```

The search gives the desired result, which indicates that the protocol can indeed be used. But the main goal of this implementation was to review any hidden assumptions in the design. After implementing the protocol in Maude, several unaddressed assumptions became clear:

- A Job is an atomic object that is translated into a single RPC call and does not depend on other Jobs or on a sequential execution of other jobs.
- The service agreement request is a part of the actual RPC service and not a standalone service.
- A client node knows the address of blackboard and server node beforehand.
- A server node knows the address of the blackboard. (The blackboard could be a local process on one of the participants to maintain complete autonomy, as in the cfengine implementation.)
- Server and client node use the same blackboard in the simulation, but this sacrifices complete autonomy by introducing a third party. (This is fixed in the cfengine implementation.)
- No explicit mention is made about the duration of the service agreement. The server will agree to serve the client indefinitely.
- A client has no way of ending a relationship with a server.
- We assume a trusted transmission medium. No lost or doubled messages can affect the result.

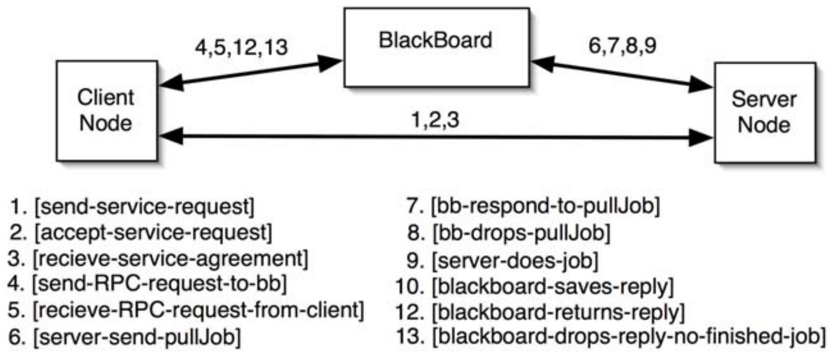
Client Node
BlackBoard
Server Node
4,5,12,13
6,7,8,9
1,2,3
1. [send-service-request]
2. [accept-service-request]
3. [recieve-service-agreement]
4. [send-RPC-request-to-bb]
5. [recieve-RPC-request-from-client]
6. [server-send-pullJob]
7. [bb-respond-to-pullJob]
8. [bb-drops-pullJob]
9. [server-does-job]
10. [blackboard-saves-reply]
12. [blackboard-returns-reply]
13. [blackboard-drops-reply-no-finished-job]

Figure 3: The different transitions in the Maude implementation of the protocol. The numbers represent the order of the transitions for a successful RPC call and reply.

Many of these assumptions are usually designed into RPC middleware, in order to provide transparency of these details.

So, after this analysis, suppose we go back to the vision of wireless mobile devices that engage with each other using a protocol like the one described. What are the ramifications to the network and service level?

Administrative Costs

A RPC call will in the Maude implementation of the protocol requires least 8 message components to be sent if there are no redundant pull messages from either side. The frequency of pull messages from both client and server is not specified in the protocol itself. A smaller interval may saturate the network, and use more battery capacity in a mobile device, but gives better service.

If we assume an equal transmission time t for a message between each participant and the pull-interval of i and a time S for a server to process the job and that all other processing of messages is close to instantaneous, we see that the smallest possible amount of time for a RPC call is: $t * 8 + S$.

Also the expected number of messages can be described as: $\frac{S}{i} + 8$.

The term *messages* is used explicitly, because it may not correspond to the a single packet per message. It may be so for the pull and service request messages, since they would be small, but for the job and the result, there are no clear limits on how small or big they may be. One should also point out, that the messages, which constitute the "overhead" compared to a direct service, are also the smallest ones. The job and the result would have to be transmitted regardless of the protocol and should not be a measure of its efficiency. What may be noted, is that they travel twice in this protocol. So for a shared medium network, they will occupy the medium twice as long.

Early in the text we pointed out the problem of current client-server technologies with regard to uncontrollable pressure on servers and the risk of bottlenecks and DoS-like saturations. Voluntary cooperation is proposed as a possible way of reducing this risk, although the negotiation phase of the protocol still contains direct traffic between the client and server. The less discussed participant in the plot, however, is the blackboard. It will receive more traffic and data then any of the two nodes that actually are using the service. If one looks at the administrative ramifications of this, one has in practice created a proxy which will get more network pressure than a typical client-server scenario would. Also, there is no voluntary cooperation in the blackboards part. Upon receiving a RPCjob it will have to store the RPCreply too at a later point.

A cfengine Implementation

The voluntary RPC has been implemented as a cfengine collaborative application. Remote services are referred to one another as "methods" or function calls. An automated negotiation mechanism has not been implemented as this remains an open question. We begin by assuming that a negotiation has taken place that determines which hosts will be clients and which will be servers and what the nature of the service is. This establishes each hosts "Method Peers," i.e., the hosts it is willing to provide services for. On the server host the update.conf rules must also contain a list of hosts to check for pending method requests.

```
MethodPeers = ( client1 client2 )
MethodPeers = ( GetPeers(*) )
```

The client *and* server hosts must then have a copy of the invitation:

```
methods:

 client_host||server_host::

 MethodExample("Service request",parameter)

 action=cf.methodtest
 server=server_host
 returnclasses=ready
 returnvars=retval

 ifelapsed=120
 expireafter=720

alerts:

 MethodExample_ready::

 "Service returned: $(MethodExample.retval)"
```

The predicate classes in the line ending in double colons tells the software that this method stanza will be read both by the client and the server host. On the client it is a request, and on the server it is the invitation. The action and server attributes determine where the methods are defined and who should execute them. Return values and classes are declared serving as access control lists for returned parameters. Since a remote method is not under out local control, we want to have reasonable checks about the information being returned. Methods could easily be asked to return certificates, for instance, to judge their authenticity. The ifelapsed and expireafter attributes apply additional scheduling policy constraints, as mentioned above.

On server host a copy of the and invitation is needed to counter-sign the agreement. The module itself must then be coded on the server. The invitation is based on the naming of the hosts; this has caused some practical problems due to the trust model. The use of IP addresses and ties to domain names has proven somewhat unreliable due to the variety of implementations of Domain Name Service resolution software. A name might be resolved into an IPv6 address on one end of a connection, but as an IPv4 address on the other side. This leads to mismatches that are somewhat annoying in the current changeover period, but which are not a weakness of the method in principle. See Display 1.

A notable point regarding this implementation is that the protocol is incorporated into an existing service. This limits the range of possible applications, but reduces extra overhead since the RPC calls and results are bundled with the already scheduled communication.

Incorporation of Voluntary RPC in cfengine

The challenge of voluntary cooperation is to balance a regularity of service (constraints on time) with the need for security against misuse attacks. The reliability of services becomes a matter for a host's reputation, rather than a matter of contract. Hosts must learn each others' probable behaviour over time rather than demand a certain level of service. Clearly this is not acceptable in every situation, but it is a desirable way of opening relations with a host that one has never seen before.

Given that all service execution is based on voluntary cooperation, there are no guarantees about service levels. This is a feature of the protocol. However, this does not mean that arbitrary bounds on service levels are not implementable. Cooperative hosts can easily arrange for a rapid level of service by agreeing on a schedule for contacting their peers. This can be as often as each party feels is appropriate.

If we assume that all host agents run diagnostics and administrative services with the same regular period P, and that all servers are willing to process the request without delay, then we can say that the expected time to service $\langle T \rangle$ is:

$$\langle P \rangle \le \langle T \rangle \le 2\,\langle P \rangle.$$

This may be verified empirically. Limits can also be placed on the maximum number of times that a method can be evaluated in a certain interval of time.

Note that the scheduling period P can be arbitrarily small, but for administrative activities one is usually talking about minutes rather than milliseconds. The efficiency of the method for small P is not impressive, due to the overheads of processing the additional steps. Thus the voluntary RPC is not suggested as a replacement for conventional service provision.

In the cfengine software implementation, the locking parameters 'ifelapsed' and 'expireafter' determine additional controls that bind the evaluation of remote methods tightly to policy. The 'ifelapsed' time says that a method will be ignored until the specified time has elapsed since the last execution. The 'expireafter' option says that a still-executing method will not be allowed to continue for longer than the expiry time.

Discussion and Comments

The implementation in cfengine does not contain a negotiation phase as a part of the RPC protocol. As implemented, cfengine assumes that type of service and its parameters is established out-of-band. Also, there is no direct reference to any blackboard that caches the RPC interaction and thus there is complete autonomy.

```
control:

 MethodName = ( MethodExample )
 MethodParameters = ( value1 value2 )
 MethodAccess = ( patterns )

 # value1 is passed to this program, so lets
 # add to it and send the result back for fun

 var1 = ( "${value1}...and get it back" )

 actionsequence = ( editfiles )

#################################################

classes:

 moduleresult = ( any )
 ready = ( any )

#################################################

editfiles:

{ /tmp/file1

AutoCreate
AppendIfNoSuchLine "Important data...$(value2)"
}

#################################################

alerts:

 moduleresult::

 ReturnVariables("${var1}")
 ReturnClasses(ready)
```

Display 1: Extended invitation example.

A more finely grained response regulation and access control is implemented in cfengine as compared to the more translucent concept of a "job" in the Maude implementation. There is little leeway for the nodes to change any of these parameters itself in any of the implementations and may therefore seem as not very dynamic or adaptable. It is also left to the implementation to sort out special cases, such as if the server would not want to serve the client but the RPC call is on the blackboard. Cfengine simply ignores such cases and clears them up as garbage. What coordination and messages would be necessary in order to remedy this situation?

The implementation in cfengine raises an important question: if this were a truly pervasive environment and the nodes just got to know each other, how would they agree on the parameters and correct return type of the RPC? Why is this important? Because if the negotiation takes more computing power than the actual service in question then what is the gain for a mobile node to participate? Seen as a game, what is the cost-benefit relationship for the client and server? One has to think of the wider context. We must try to keep in mind future services that nodes may offer to each other, where such a negotiation is worth the delays.

Based on our observations, it is natural to think of a realistic future scenario as being partially mobile, perhaps with some fixed infrastructure. that may serve as a cache for blackboard messages. For example, in a virtual shopping mall, a blackboard service might be provided, taking its own share of the risk from customers. Nodes have greater freedom through this protocol: they may actually be away from the network while the contents are cached by the blackboard. A global ID, like the MIPv6 protocols home address, will make sure that the nodes can gather the data belonging to it from anywhere. It must be noted also, that the blackboard does not have to be a single hop away. It can be several, making this service more global and not just limited to a single network.

So, is there really any need for this type of voluntary cooperation?

One model of pervasive services is commercial and based on the Internet café. Here roaming tourists pay for a ticket that grants them access to an otherwise closed local service. In regular commerce, it is assumed that this monetary relationship is sufficient to guarantee a decent form of behaviour, in which parties behave predictably and responsibly. But is this necessarily true? There are several faults in this assumption for network services.

First of all, the commerce model assumes that virtual customers will behave much as they have done in human to human commerce. There is no compelling reason to suppose that this will be the case. A generation of users has now grown up taking lawless online technologies for granted. Technologies such as mobile phones, peer to peer sharing and Internet chat rooms are *changing* societal attitudes and the rules of interaction between young people [29].

Secondly, the model assumes that clients will exhibit predictable behaviour. Studies show that such a predictable relationship must be built up over time, and only applies if customers are regular patrons of a service provider [30]. Customer trust is based on the idea that customers and providers form relatively long lasting relationships and that one can therefore punish misdeeds by some kind of tit-for-tat interaction [8], because they will almost certainly meet again in the future. However, in a future where everything is on the move, distance has no meaning, and both children and adults are very much on-line, there is reason to suppose that clients will not 'hit and run' servers for prank or for malice. Hiding or changing identity on the Internet is trivial, so customers can visit providers repeatedly with a new identity and there avoid reprisals.

Finally, the use of of immediate payment as a barrier to deter frivolous usage assumes users will behave rationally in an adult fashion. This assumption is also flawed, since users might expect their uncooperative actions to have greater payoffs down the line. All of this points to the need for greater initial caution, and a reward scheme for cooperative behaviour.

A simple protocol for voluntary cooperation has shown that with a slightly higher load on a network one can reduce the risks of connecting with untrusted clients. If these types of services arrive, using independent blackboards (e.g., Grid Computing), the system administrators may see the dawn of a new type of servers which act only as caches and proxies for voluntary interaction. They will in turn be the ones that need protection from saturation and abuse. If fully autonomous behaviour is preserved, these problems can be avoided. We are currently working on a theory of fully autonomous interaction called promise theory [31].

Is voluntary cooperation a technical challenge or a human challenge then? The answer must be that it is both. A device that is misappropriated for malicious purposes by a human behaves simply as a hostile device to other computers. Computers perform a useful service to maintain the integrity of the networked community if they try to protect themselves, regardless of the source of the mischief.

Conclusions

This paper discusses the issue of voluntary cooperation in peer networks, and analyses a proposed trade-off which might help to lower the risks of client-server interactions in unpredictable environments.

The mechanism proposed is one of minimal acceptable trust. Through the mechanism, hosts in a collective environment can maintain maximal administrative control of their own resources, and hence reduce the risk of abuses and security breaches. Voluntary cooperation (or pull-based service provision) is fully implementable and offers flexibility of asynchronous timing, as well as some additional security.

The motivation for this work was initially to support an administrative peer to peer algorithm that respects the autonomy of individual peers (inspired by model 6 of refs. [13, 3]), but the result is of more general interest. Its application domain is mainly tasks that are the traditional domain of the system administrator, but which have to operate in an uncertain and far from controlled environment.

A simulation in the analysis framework Maude, and an test implementation in the configuration tool cfengine [32, 33], show that the protocol functions properly, as advertised, and reveals the nature of the tradeoffs. The protocol overhead can be reduced, if voluntary cooperation is incorporated into an existing service, as in cfengine.

We have tested the voluntary cooperation approach on mundane tasks like the backup of remote machines, distributed monitoring (exchange of usage data, confirmation of uptime etc.), load balancing and other tasks. Although this mechanism currently has a limited number of desirable applications, this is probably due, in part, to the lack of truly mobile administrative approaches to hosts, in contemporary organizations. As pervasive services mature, one could expect that a voluntary cooperation mechanism would actually become invaluable for high risk transfers, e.g., in distributed software updating.

For demand services, the minimal trust model seems both cynical and inappropriate: perhaps even a hindrance to cooperative behaviour and commerce. At some point, the actors of the virtual community online must learn the human rules of trusting engagement, and new generations of children will have to be educated to accept that virtual communities are just as important as real ones. Human relations are based on rules of voluntary etiquette, but such social niceties have taken thousands of years to evolve.

We present this work in the framework of system administration because system administrators are the virtual custodians of the hosts and devices in the pervasive computing scenario. If administrators do not discuss the issues of scalable and management, and devise a workable scenario in the face of all its uncertainties, no one will.

Acknowledgement

We are remotely grateful to Siri Fagernes and John Sechrest for voluntary discussions on cooperative services.

Bibliography

[1] Sapuntzakis, C. and M. S. Lam, "Virtual appliances in the collective: A road to hassle-free computing," *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HOTOS IX)*, 2003.

[2] Begnum, K., M. Burgess, and J. Sechrest, "Infrastructure in a virtual grid landscape from abstract roles," *Journal of Network and Systems Management*, (submitted).

[3] Burgess, M., and G. Canright, "Scaling behaviour of peer configuration in logically ad hoc networks," *IEEE eTransactions on Network and Service Management*, Vol. 1, Num. 1, 2004.

[4] Burgess, M., and S. Fagernes, "Pervasive computing management: A model of network policy with local autonomy," *IEEE eTransactions on Network and Service Management*, submitted.

[5] Burgess, M., and S. Fagernes, "Pervasive Computing Management II: Voluntary Cooperation," *IEEE eTransactions on Network and Service Management*, submitted.

[6] Burgess, M., and S. Fagernes, "Pervasive computing management III: Management Analysis," *IEEE eTransactions on Network and Service Management*, submitted.

[7] Undercoffer, J., F. Perich, A. Cedilnik, L. Kagal, and A. Joshi, "A secure infrastructure for service discovery and access in pervasive mputing," *Mobile Networks and Applications*, Vol. 8, Num. 2, pp. 113-125, 2003.

[8] Axelrod, R., *The Evolution of Co-operation*, Penguin Books, 1984.

[9] Martin-Flatin, J. P., "Push vs. pull in web-based network management," *Proceedings of the VI IFIP/IEEE IM conference on network management*, p. 3, 1999.

[10] Burgess, M., "Computer Immunology," *Proceedings of the Twelth Systems Administration Conference (LISA XII)*, p. 283, USENIX Association, Berkeley, CA, 1998.

[11] Paxson, V. and S. Floyd, "Wide area traffic: the failure of poisson modelling," *IEEE/ACM Transactions on networking*, Vol. 3, Num. 3, p. 226, 1995.

[12] Bellavista, P., A. Corradi, R.Montanari, and C. Stefanelli, "Policy-driven binding to information resources in mobility-enabled scenarios," *Mobile Data Management, Lecture Notes in Computer Science*, Num. 2574, pp. 212-229, 2003.

[13] Burgess, M., and G. Canright, "Scalability of peer configuration management in partially reliable networks," *Proceedings of the VIII IFIP/IEEE IM conference on network management*, p. 293, 2003.

[14] Anderson, E., M. Burgess, and A. Couch, *Selected Papers in Network and System Administration*, J. Wiley & Sons, Chichester, 2001.

[15] Fradet, P., V. Issarny, and S. Rouvrais, "Analyzing non-functional properties of mobile agents,"

Fundamental Approaches to Software Engineering, Lecture Notes on Computer Science, Num. 1783, pp. 319-333, 2000.

[16] Case, J., M. Fedor, M. Schoffstall, and J. Davin, "The simple network management protocol," *RFC1155*, STD 16, 1990.

[17] Zapf, M., K. Herrmann, K. Geihs, and J. Wolfang, "Decentralized snmp management with mobile agents," *Proceedings of the VI IFIP/IEEE IM conference on network management*, p. 623, 1999.

[18] Matsushita, M., "Telecommunication management network," *NTT Review*, Num. 3, pp. 117-122, 1991.

[19] Sloman, M. S. and J. Moffet, "Policy hierarchies for distributed systems management," *Journal of Network and System Management*, Vol. 11, Num. 9, p. 1404, 1993.

[20] Lignau, Anselm, Jürgen Berghoff, Oswald Drobnik, and Christian Mönch, "Agent-based configuration management of distributed applications," *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 52-59, 1996.

[21] Oram, Andy, editor, *Peer-to-peer: Harnessing the Power of Disruptive Technologies*, O'Reilly, Sebastopol, California, 2001.

[22] Dunbar, R., *Grooming, Gossip and the Evolution of Language*, Faber and Faber, London, 1996.

[23] Lee, M. K., and X. H. Jia, "A reliable asynchronous rpc architecture for wireless networks," *Computer Commmunications*, Vol. 25, Num. 17, pp. 1631-1639, 2002.

[24] Kottmann, D., R. Wittmann, and M. Posur, "Delegating remote operation execution in a mobile computing environment," *Mobile Networks and Applications*, Vol. 1, Num. 4, pp. 387-397, December, 1996.

[25] Bakre, Ajay V., and B. R. Badrinath, "Reworking the RPC paradigm for mobile clients," *Mobile Networks and Applications*, Vol. 4, pp. 371-385, 1997.

[26] Burgess, M., G. Canright, and K. Engø, "A graph theoretical model of computer security: from file access to social engineering," *International Journal of Information Security*, Vol. 3, pp. 70-85, 2004.

[27] Watts, D. J., *Small Worlds*, Princeton University Press, Princeton, 1999.

[28] Maude, *The maude homepage*, http://maude.cs.uiuc.edu .

[29] Rheingold, Howard, *Smart Mobs: The Next Social Revolution*, Perseus Books, 2002.

[30] Axelrod, R., *The Complexity of Cooperation: Agent-based Models of Competition and Collaboration*, Princeton Studies in Complexity, Princeton, 1997.

[31] Burgess, Mark, "An approach to understanding policy based on autonomy and voluntary cooperation," *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97-108.

[32] Burgess, M., "A site configuration engine," *Computing systems*, Vol. 8, p. 309, MIT Press, Cambridge, MA, 1995.

[33] Burgess, M., *Cfengine www site*, http://www.iu.hio.no/cfengine, 1993.