Component Selection for Heterogeneous Active Networking

Steven Simpson, Mark Banfield, Paul Smith, and David Hutchison

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{ss, banfield, smithp1, dh}@comp.lancs.ac.uk

Abstract. Active Networking (AN) involves the processing of programs in heterogeneous networking environments. There are several AN solutions, exposing different APIs and using different languages, and each may be appropriate for different tasks such as high-speed multimedia processing or low-speed routing adjustments.
We describe our active node system, LANode, that separates control- and data-plane activities, and introduce Component Compatibility Markup Language (CCML), a critical component of LANode that allows it to be applied to heterogeneous platforms.

1 Introduction

Active Networking (AN) is an approach to providing new network services without major changes to network infrastructure or formal standardisation. Instead, code is dynamically installed into network nodes to replace or augment their basic function of routing or switching [18]. Areas of AN research include code-installation methods, portability and security versus functionality in the choice of language or programming environment, and packet delivery (virtual networks versus packet interception).

At Lancaster, we are developing LANode, an active platform which can be adapted for, and deployed on, heterogeneous hardware, operating systems and AN environments. LANode can provide node-specific services, and active applications on LANode adapt themselves to these services to expose their own node-independent services. We deal with node heterogeneity using a form of content negotiation involving an XML document format, CCML (Component Compatibility Markup Language).

We report on recent AN technologies in Sect. 2. LANode and CCML are detailed in Sects. 3 and 4. Our planned developments and research activities are described in 5.

2 Active Networking

The aim of active networking is to allow new services to be dynamically deployed in a network without changing the core functionality of the network components or consulting standardisation bodies. Programs to support the services are loaded into the network nodes to process traffic intended for them, with varying levels of dynamism, and of restriction by the network operators.

2.1 AN Characteristics

Several AN architectures and environments have been proposed and developed. We now describe some of the various dimensions of these systems.

Program installation In the *discrete* approach, programs are loaded onto nodes that particular traffic is anticipated to pass through. In the *in-band* (or *capsule-based*) approach, programs exist (or are referred to) within the traffic, and loaded as required.

Planes Network services may process traffic in various ways. Some may process large volumes to control network load (e.g. as filtering does) or to service heterogeneous clients (e.g. transcoding) — we describe these as *data-plane* activities. Other services may process control information that subsequently affects larger volumes of traffic (e.g. routing), or may gather the information about such traffic — these are *control-plane* activities.

Different programming environments are suited to different activities. A machine-code, zero-copy, kernel-space environment is more appropriate for high-speed, data-plane activities, whereas high-level languages with extensive data abstractions and libraries are better for sophisticated control-plane activities.

Integrity Safety and security are vital if a network's hardware is to be exposed to programs performing unforeseen activities. These may be achieved through use of restricted programming languages, access-controlled libraries (the sand-box model), and formally verified programs.

Traffic capture AN environments must deliver traffic to their installed programs by some means. Traditionally, packets are *generated* within the active network, to be passed across a virtual network overlaying a conventional network, but packets may also be intercepted as they traverse a node using conventional protocols. In the latter case, the environment must provide facilities for specifying which packets to intercept (e.g. Berkeley Packet Filtering (BPF) [14]), and due to the potential overhead of having large numbers of complex packet filters, approaches have been sought to perform efficient multi-field classification [11].

2.2 AN Work

Much of the early work on AN was carried out as part of an American Department of Defense (DARPA) initiative to develop network technologies that were highly adaptable and robust. Some of the results from this early work are summarised in [17]. An overview of some of the significant AN platform research follows.

Deployment and Experimentation The ABONE is a shared virtual network of nodes for conducting AN large-scale research [1]. Independent developers can provide their own *execution environments* (EEs) within which *active applications* (AAs) can run. EEs are manually deployed across ABONE nodes using a common management interface on each node, and AAs are loaded into an EE according to its specification. Packets traversing the ABONE are encapsulated using ANEP, which specifies the intended EE type by a global numeric identifier, so different EE developers can run experiments independently.

Execution Environments The ANTS toolkit [18], developed at the Massachusetts Institute of Technology, is one of the earliest approaches to providing active technologies in the network. It adopts the in-band or capsule approach to program installation, where a packet includes a reference to a forwarding routine, which is used to process the packet at each ANTS node. Java is the programming language in which capsules are programmed. This enables the capsules to be executed in a safe sand-boxed environment.

The SwitchWare architecture [2] is another of the early active architectures to emerge from DARPA-funded research. SwitchWare packets contain both code, in the form of PLAN (Programming Language for Active Network) statements, and data. In addition to active code embedded in packets, active extensions (commonly used code resident on a node) enable a greater level of functionality and can be referenced from PLAN. Security in SwitchWare is provided using a combination of both cryptographic based authentication and formally verifiable programming languages.

Active/Programmable Packet Processing Unless active packets are directly addressed to the next active node, nodes must provide mechanisms for efficient packet capture and processing.

LARA++ [16] develops work carried out on LARA (Lancaster Active Router Architecture) [5]. A component based architecture has been adopted enabling a greater degree of flexibility than first generation active node architectures. In addition to this, active components execute in a zero-copy, user-space environment with a minimal performance penalty. LARA++ aims to make the development of active programs easier because of these properties, without a significant performance hit.

The Router Plugins [7] active router architecture is oriented around the concept of a plugin — the base unit of modularity in the architecture that performs some form of meaningful computation. Plugins can be dynamically loaded at run time and bound to specific flows to perform computation on those flows.

Pronto [12] is intended to be a platform on which high level AN research can be trivially conducted. This is due to the fact that Pronto is based upon and extends the functionality of a commodity OS (Linux). Pronto is intended to support multiple heterogeneous execution environments. In a similar manner to LANode (see Sect. 3), attention has been paid to the identification and separation of the service specific and service generic facilities of the platform and the interfaces between them.

Application-Level Active Networking The University of Technology, Sydney has taken an alternative approach to active networking by moving it into the application level [9]. They introduce the general concept of *Application-Level Active Networking* (ALAN), which involves dynamic installation of proxy services on suitable nodes. Active capsules come in the form of *proxylets* which can execute in an *Execution Environment for Proxylets* (EEP; historically also *Dynamic Proxylet Server*, DPS).

UTS have defined a Java API that distinguishes between an initialisation phase and a daemon phase of the proxylet's execution. They have also built an EEP implementation in Java called *funnelWeb*, and have constructed a virtual network of these for testing and evaluation of funnelWeb, and of various distributed algorithms running across proxylets [10]. The use of Java ensures portability, and provides the security mechanism.

funnelWeb has been used to provide multicast bridges to connect MBONE islands, TCP bridges to route around poor international connections, HTTP-to-RTP gateways to extend the functionality of web servers.

Current work with the EEP is focused on providing application-level routing so that proxylets can co-operate by building meshes of communication according to an application's criteria [10].

3 LANode

ALAN consists of proxylets performing application-dependent activities in a Java environment. Applications must be explicitly configured to use them, such as a browser being configured to use a HTTP proxy. Also, the use of Java for programming proxylets ensures widespread compatibility, but can be slow for high-volume data-plane activities, particularly if packets are copied into user space.

We have chosen to extend the ALAN/proxylet approach in such a way that the advantages of portability and security can be retained, while the use of machine-optimised code and kernel-space packet processing are introduced. LANode allows proxylets to perform data-plane tasks efficiently, and without being directly addressed by the applications using them.

3.1 Architecture

LANode is a development of the ALAN EEP, and consists of two planes (as in Fig. 1):

Control plane This offers a conventional EEP environment to run Java proxylets that provide mainly control and management functions.

Data plane This may offer various execution environments, including ones specific to processor or operating system type. Optimised for high-performance packet processing, most network traffic is expected to traverse this plane, and avoid the control plane.

This is a common distinction found in operating systems that have both a general use and an IP forwarding function: packets traversing the node never

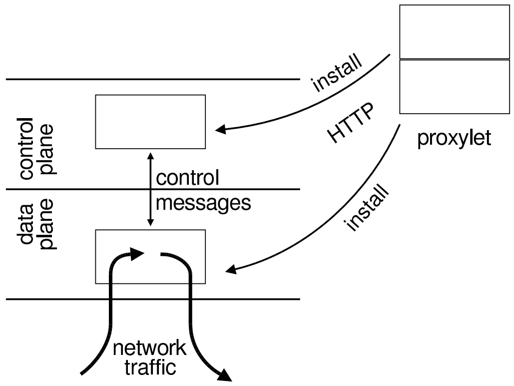

Fig. 1. LANode EEP control and data planes

leave the kernel (the data plane), but utilities can modify the routing table from user space (the control plane). Processing in the control and data planes run independently of each other, except for occasional interaction between the them when the data plane is reconfigured, or the control plane is informed of activity.

The EEP API is extended to provide access to the node's resources through a set of named services called *profiles*. This set can vary from one node to the next, according to what it can best provide – we do not prescribe a single, broad, fixed interface.

Some profiles provide adminstrative services such as a CORBA naming service or an RMI registry so that proxylets can present their own named services. Other profiles expose the data plane to configuration with varying flexibility. For example, exposing the node's IP routing table allows some control over traffic. Greater flexibility can be obtained through a profile that allows active code to be installed in the data plane, e.g. a node on a Linux system may have a profile that allows the installation of kernel modules. Once in place, the two parts can interact through the profile, usually to exchange configuration information.

Each profile has a name, a Java class/interface type, and documented behaviour, ensuring that if a profile is provided on two different nodes, one can be certain that they behave identically. However, it is not necessary for every node to support every profile. The total set of profiles can be expanded as new services are foreseen (though clearly, a minimal set is desirable), and the naming scheme is hierarchical so that organisations can develop profiles independently.

A particular node will be configured with properties describing the characteristics of the node, including a list of the available profiles. Other properties may indicate the operating system or the processor type. For example, a node may specify that its data-plane environment is for Linux kernel modules consisting of i386 code, and along with netfilters[1] support, and that there is a profile

[1] The netfilters kernel extension allows convenient packet interception in later Linux kernel versions.

to manipulate the IP routing table. This information allows a proxylet to be optimised at install time for the node's characteristics, including the type of data-plane environment provided.

A proxylet needs to ensure that the code it loads into the data plane is compatible with the environment there. One approach would be to allow a loaded proxylet to interrogate LANode to determine its capabilities to which the proxylet adapts. Instead, LANode adapts the proxylet before loading, through choice of components and installation parameters. We have chosen to allow the proxylet to be composed of several network-available components at installation time. The proxylet can be expressed as a CCML document (see Sect. 4) that lists all the possible components, and indicates the LANode characteristics (such as profile, processor type, operating system) that each component is compatible with, and it is this document that is submitted to the node to install a proxylet. The node compares its characteristics with the document, and selects a set of components from which to build proxylet.

This selection allows, for example, one compilation of a Linux kernel module to be selected from compilations for several processor types. Independently, part of a proxylet may be an RMI object or a CORBA object, depending on whether an RMI registry or a CORBA name service are locally available.

3.2 Implementation

LANode is a combination of two parts:

- The core provides the extended EEP environment, along with the basic facilities for loading, starting and stopping a proxylet through a CORBA interface.
- The stub provides the profile implementations, and is supplied to the core as run-time configuration.

This approach separates the development of the EEP and the profiles. For example, our current core spawns a virtual machine (VM) for each proxylet, but a future version may manage all of them in a single VM. This choice is independent of the provision of profiles and other node characteristics, so the same stub should be compatible with both cores.

We have built a stub designed for Linux, providing profiles to allow:

- access to an RMI-like CORBA registry for exposing services provided by proxylets,
- access to the node's IPv4 routing table,
- modules to be loaded into the kernel, and contacted through a virtual device,
- shared libraries to be linked to the virtual machine to support native methods, and
- executable programs to be installed and used as required.

The Java security mechanism is employed to check the validity of any attempts to use these profiles. For example, loading modules, libraries or executable programs requires a `NativePermission` that specifies where the code can be loaded from, and for what purpose.

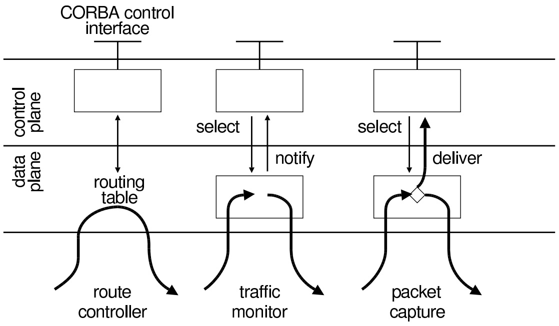

Fig. 2. Developed proxylets

3.3 Applications

By exposing the data plane to LANode proxylets, we aim to provide proxylets optimised for various platforms providing high-performance packet processing, while exposing interfaces (CORBA/RMI) to sophisticated control objects to adjust that processing. The same proxylet can be loaded onto several heterogeneous nodes, and perform the same task efficiently.

Proxylets developed so far (depicted in Fig. 2) include:

– an IPv4 routing proxylet, exposing a routing table through a CORBA interface,
– a traffic detection proxylet, exposing a CORBA interface to register an interest in packets traversing a node,
– a packet diversion proxylet to allow packets belonging to particular flows to be intercepted and processed in the control plane[2],
– a transparent proxying proxylet (depicted in Fig. 5) which can intercept a TCP connection to a given host and port, and redirect it to an alternative host and port.

4 CCML

LANode is intended to be deployed on heterogeneous nodes, and must be prepared to run active code on any type of system, and it is assumed that a particular active application has been generated several times, once for each system type. (LANode permits a proxylet to consist of several parts with varying degrees of system dependence, so the generic parts only need to be generated once.) So LANode requires a mechanism for selecting one of these versions which is compatible with its node's characteristics. This function is analogous to content negotiation, as used in the WWW, and supported through HTTP [8].

[2] This is intended for applications where the amount of traffic to be processed is quite small.

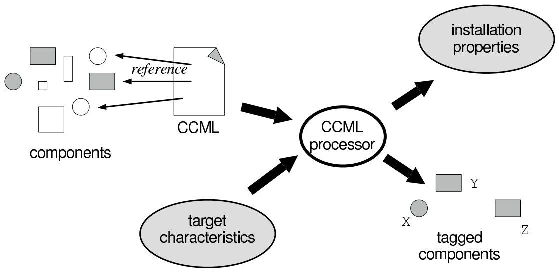

Fig. 3. CCML function

There are two main approaches to content negotiation: agent-driven and server-driven.

agent-driven An ordinary request is met with a response listing several alternatives and their characteristics. The requesting agent must choose items that are compatible with its own characteristics, and make further requests.

server-driven The request is augmented with the agent's own requirements, and the server chooses an appropriate response, and returns it directly.

The latter involves only a single interaction, but is more difficult to cache in proxies. The former works better with caches, but requires at least one extra interaction. Whether this is significant in the case of LANode (which has the role of the agent) is an issue for further study.

Component Compatibility Markup Language (CCML) is an XML [4] format to represent responses during agent-driven content negotiation. Instead of the agent (e.g. LANode) specifying its own characteristics in a request for an entity (e.g. a proxylet), it requests the entity's CCML document. The agent should be able to resolve the document against its own characteristics, which produces information about a version of the entity that is tailored toward the agent. It is not limited to resolving to a single reference — the resultant components are expected to be (re-)composed by the agent to form the resource.

An agent can combine a CCML document with its own 'target characteristics' (name-value strings), as in Fig. 3, to produce a set of compatible components, plus some installation properties (name-value strings). Additionally, each component may have an attached parameter (just a string).

4.1 Related Work

Component selection or content negotiation also exists in other languages. SMIL is intended for media components to produce multimedia presentations by specifying their relative spacial and temporal positions [3]. It also allows alternative media to be presented based on client characteristics such as preferred natural language, or bandwidth available to the client. This function, provided by SMIL's switch element type, is equivalent to CCML's function. However, the names of testable characteristics are expressed directly as attribute names, rather than attribute values, so adding new characteristics (or replacing them entirely) for an alternative application requires the DTD to be abandoned. Also, the mathematical relationship ('equals', 'greater than'. . .) between a characteristic and a given value is implied by the name, whereas CCML allows this to be expressed as an attribute value, in order to be more independent of its application.

For active networks, XML has already been applied to specifying the semantics and non-functional characteristics of distributed objects, so that collaborative applications can be built with predictable behavioural characteristics [15].

4.2 Syntax and Semantics

The syntax of CCML is defined by the XML DTD at [6]. A CCML document consists of a sequence of 'actions', many of which may be conditional. Conditions are expressed through 'expression' elements, with comparisons against platform characteristics forming the primitive expressions, and logical operators forming compound expressions.

Processing a CCML document begins with initialising state: an empty list of tagged component references, and an empty set of properties (to become the 'installation parameters' in Fig. 3). Each action element is then processed in sequence, unless an enclosing element somehow precludes it. Processing an action may result in:

- an item in the installation parameters being set, reset or overwritten,
- a component reference being added to the list, with or without a tag,
- a previously added reference being tagged.

Expression elements consist of:

- `REL`, representing boolean relationships between configuration properties and supplied values,
- `LOGIC`, representing multi-operand logical expressions, and containing those operands,
- `EXT`, refering to other expressions in the current document or others,

while action elements consist of:

- `LOAD`, which specifies the (relative) URI of a component to be added to the component list,
- `STORE`, which associates a tag with a component URI,
- `SET`, which sets the value of a global property or installation parameter,
- `BRANCH`, which contains some expressions, followed by actions only to be performed if the expressions are all true (it is said to be compatible),
- `DECIDE`, which contains only branches, only the first of which is compatible is selected.

Expressions may also appear outside `BRANCH` elements, where only actions are normally expected. These are ignored except when referenced by an `EXT` element.

It is the `DECIDE` element type that allows alternative components to be selected. The actions within only one of its branches are processed, and if no branches are compatible, the processing is aborted.

4.3 LANode Application of CCML

LANode contains a CCML processor, and accepts proxylets either as single JAR files, or as CCML documents. In the latter case, the document is processed according to the LANode's properties, and the resultant list of components is used to form the proxylet's class path (the tags are ignored). Furthermore, the derived installation parameters are made available to the proxylet as its configuration.

If the CCML document cannot be fully processed, e.g. because there are no compatible combinations of the proxylet components, the installation fails.

LANode Selection Criteria To illustrate the use of CCML in LANode, we describe some typical properties that a LANode node may provide.

profile lists profiles supported by the node.
arch lists processor types supported in the data plane.

These are not yet fixed, since it is not clear whether a node should be permitted to provide several similar variants of a service, particularly when there is more than one dimension of variance (e.g. loading a Linux kernel module on two processor types, and with two different sets of kernel extensions). This is a topic of further study.

Depending on the values of the properties above, other criteria may be present, possibly refining their meaning. For example:

linux.kernel-environment lists extensions to the Linux kernel, such as netfilters.

LANode Profiles As with the selection criteria, these allocations are yet to be fixed. In particular, the set of profile types is expected to grow as new services with nodes are defined and exploited.

nativecode.CharDeviceLoader permits loading of data-plane modules that provide a virtual character device, accessible from user space, for control purposes.
nativecode.ProgramLoader permits loading of executable programs which can be accessed in Java through a `Process` object.
registry.Registry permits access to a registry of local CORBA objects which proxylets have provided.

Profile names are closely associated with the names of their interface types, e.g. CharDeviceLoader has a Java interface type of `CharDeviceLoader`. While this association is desirable, it is not a requirement.

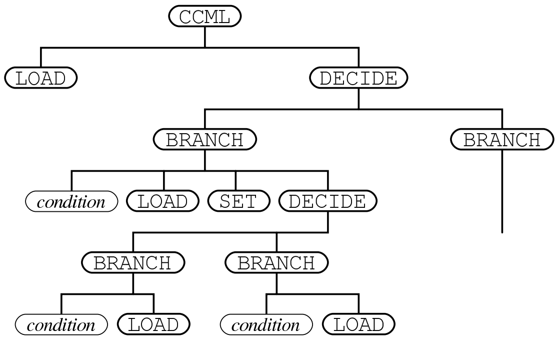

Fig. 4. Graphical structure of CCML

Adaptable Proxylets Given the above LANode properties and profiles, we can construct a proxylet suitable for a wide variety of platform types by separating its functionality according to level of platform dependence.

For the proxylet that performs transparent proxying, we can isolate several platform-independent parts – external control-interface CORBA stubs, a veneer interface (the veneer will adapt the node's profiles to the transparent proxying service), and a main function to bind them together – and specify them unconditionally in a CCML document with several `<LOAD>` elements.

Then we can produce several veneer implementation classes, one for each useful profile (e.g. the Linux kernel module loader) — LANode only needs to select one of these, so they are listed as alternaive `<BRANCH>`es in a `<DECIDE>`. We can develop new classes as new profiles are defined. As well as identifying the appropriate JAR file with `<LOAD>`, CCML can be used to pass the name of the class within it that should be used as the implementation.

For each of these implementations, there may be further dependencies — the module loader may be supplied with a module chosen from several precompiled alternatives for different processor types.

This branching and refinement at each level is depicted in Fig. 4.

4.4 Example

The current implementation of our Linux-based LANode stub together with the transparent-proxying proxylet serve as an example use of CCML. Firstly, the proxylet is broken into several components as JAR files:

- `iptpidl.jar` — CORBA IDL stubs for the control interface,
- `iptpproxylet.jar` — core proxylet implementation and CORBA control implementation,
- `iptpv.jar` — veneer interface to control transparent proxying,
- `iptpcdev.jar` — veneer implementation based on a loadable virtual character device,

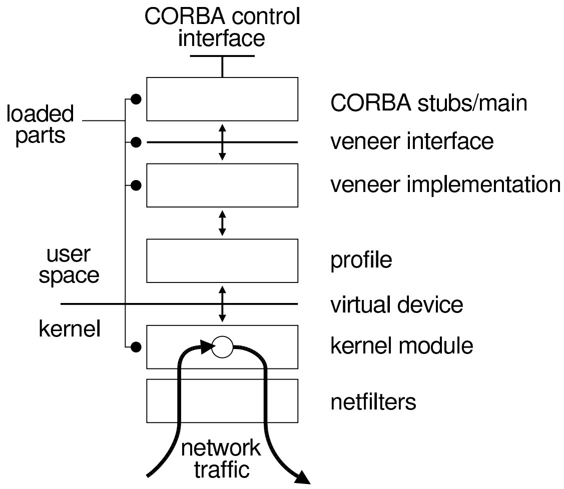

Fig. 5. Interactions between proxylet components in LANode/Linux

– `iptp-linux-i386.jar` — transparent-proxying module for a Linux kernel, compiled for i386 processors.

These will form the running proxylet, and interact as in Fig. 5. Control information passes from the CORBA interface, through the the veneer, the profile and the virtual device, and alters the behaviour the module and the traffic passing through it.

The CCML document would include the fragment in Fig. 6. The first three components are generic components, so they appear unconditionally in the CCML as `<LOAD>` elements.

`iptpcdev.jar` adapts the CharDeviceLoader profile to the veneer interface – in practise, this means translating calls to the interface methods into character streams to be transmitted through the virtual device. It must only be used if that profile is available, i.e. character devices can be loaded into the data plane, so its entry in CCML is preceded by a `<REL>` condition expressing this requirement, and placed in a `<BRANCH>` element. (If there were other alternatives (as in a richer scenario), they would appear as other branches in a `<DECIDE>` element, which is also shown in the example.) The component is also accompanied by an installation parameter (a `<SET>` element) to tell the generic parts of the proxylet which class from `iptpcdev.jar` implements the veneer interface.

Having selected `iptpcdev.jar`, it must have a component to load into the kernel. Only one is provided in this limited example, and this is the file `iptp.o` in `iptp-linux-i386.jar`. This forms part of the proxylet only if the data plane is a Linux kernel on an i386 processor — these conditions are placed within a `BRANCH` element, and an installation parameter is included to tell the veneer implementation where to find the module. The branch is placed within a `DECIDE`

```
<!-- generic components --> <LOAD URI="iptpproxylet.jar" />
<LOAD URI="iptpidl.jar" />  <LOAD URI="iptpv.jar" />
<DECIDE>
  <BRANCH> <!-- only for character-device data planes -->
    <REL PROP="profile"
          ARG="UK.ac.lancs.nativecode.CharDeviceLoader" />
    <SET NAME="veneerName" VALUE=
"UK.ac.lancs.iptp.chardev.CharDeviceIPv4TransProxyFactory" />
    <LOAD URI="iptpcdev.jar" />
    <DECIDE>
      <BRANCH> <!-- only for i386 Linux data planes -->
        <REL PROP="arch" ARG="i386" />
        <REL PROP="os" ARG="linux" />
        <LOAD URI="iptp-linux-i386.jar" />
        <SET NAME="UK.ac.lancs.iptp.IPv4DeviceImpl"
             VALUE="iptp.o" />
      </BRANCH>
      <!-- further alternatives... -->
    </DECIDE>
  </BRANCH>
</DECIDE>
```

Fig. 6. Example CCML

```
profile=UK.ac.lancs.nativecode.LibraryLoader,
        UK.ac.lancs.nativecode.ProgramLoader,
        UK.ac.lancs.nativecode.ModuleLoader,
        UK.ac.lancs.nativecode.CharDeviceLoader,
        org.ieee.p1520.ipswg.routing.RoutingTable,
        UK.ac.lancs.registry.Registry
os=linux
arch=i686,i586,i486,i386
```

Fig. 7. Example LANode selection properties

element alongside the character-device veneer implementation. If other processor types are supported by the proxylet, they will have similar branches inside the `DECIDE`.

LANode with the Linux stub has the characteristics in Fig. 7. Applying these to the CCML document produces a list of all the components listed above, plus the installation parameters of Fig. 8.

On a LANode with a different configuration, the document will not resolve, and an error will be reported. As compatible components for alternative platforms are written, they can be added to the document, making the proxylet as a whole compatible.

```
veneerName=
  UK.ac.lancs.iptp.chardev.CharDeviceIPv4TransProxyFactory
UK.ac.lancs.iptp.IPv4DeviceImpl=iptp.o
```

Fig. 8. Example installation parameters

```
import java.util.*;
import java.net.*
import UK.ac.lancs.ccml.*;

// inputs
URL documentLocation = ... ;
Properties selectionProperties = ... ;

// outputs
Properties conf;
ComponentTransfer[] transfers;

ComponentSelector selector =
  new PropertiesComponentSelector(selectionProperties);
ComponentURLResolver resolver =
  new ComponentURLResolver(selector);

conf = new Properties();
transfers = resolver.resolveToTransfers(documentLocation, conf);
```

Fig. 9. Java code for CCML processing

4.5 CCML Processing

We have constructed a Java library to perform evaluation of a CCML document against supplied properties, producing a list of tagged URIs and some configuration properties.

This library forms the 'CCML processor' component of Fig. 3. In the code fragment of Fig. 9, `documentLocation` is a reference to the CCML document, and `selectionProperties` corresponds to the 'target characteristics'. After processing, `conf` holds the 'installation properties', while `transfers` lists the component references and their tags.

5 Future Work

5.1 LANode Developments

We intend to develop and settle the extended EEP API so that all future development is reduced to the definition of profiles. One such profile could present a generic router abstraction consisting of various components to classify, modify or forward packets through the data plane, and it should be possible to add further components dynamically. This might be based on the Click modular router [13].

We are investigating ways to allow efficient traversal of these components without the overheads of matching packet fields to configured values, e.g. by hashing on several fields (as used in [7]) and determining if a given classifier is based solely on those fields, or by employing a more generic, efficient, multi-field method of classification, e.g. [11].

5.2 CCML Developments

CCML should have other applications, and in some cases, the language may have to be extended for a particular task. It already permits selection of components used to compose a proxylet, and should also be applicable to:

- general software installation, whereby versions of software packages and plugins appropriate to a system can be installed there using a single reference to the software,
- layered multimedia delivery, in which, say, an image is hierarchically decomposed according to various dimensions (colour, spatial resolution, cropping position), and presented through HTTP as a CCML document that allows the components to be minimally selected according to the characteristics of the rendition,
- stylesheet selection, where the dependence of the suitability of a stylesheet for a given medium can be more richly expressed.

We will also investigate the use of CCML as a response to a URN resolution request. This would reduce a name registration to the storage of a static document which is then resolved by the client (in terms of its own properties) after receiving the response. In cases where the document is very large, the client may submit some of its properties with the request so that a partially resolved CCML document is returned.

It may be necessary to extend CCML to cope with property types more closely associated with name resolution, for example, global position. These types could be added statically to the language, or we may devise a method of describing types through other documents to be processed dynamically.

There may be problems if CCML is applied to situations where several independent dimensions of compatibility exist, resulting in $m \times n$ alternative components that must somehow be expressed as CCML elements.

Future CCML developments will appear at our website [6].

6 Conclusions

We have presented LANode, an active node that employs the discrete approach to active-code deployment. It recognises the distinction between control-plane and data-plane behaviour, and allows active code (proxylets) to span both planes.

LANode has a small core 'API', but may have various extensions (profiles) per node, depending on the existence of programmable entities in the underlying operating system and hardware. This allows it to be deployed across networks of heterogeneous platforms, and to develop as new platforms appear and others fall into disuse.

LANode's proxylets may be built from several components to be composed when they are deployed. Some components will be independent of a node's available profiles, and so will be used in all deployments of a proxylet; others will be profile-dependent, conditionally linked with the proxylet to allow it to adapt to the available profiles with each deployment. These components may be re-used in other proxylets that need to perform the same kind of adaptation.

We have presented CCML, a document format which expresses how a proxylet is to be formed from potential components depending on a node's profiles and other properties; wholely incompatible proxylets can be rejected before any code is loaded. As new platforms appear, and profiles for them are defined, an existing proxylet can be extended to make use of them (and therefore adapt to the new platforms) by writing a new component and altering the proxylet's CCML document. CCML may have other uses in content negotiation.

Acknowledgments

We gratefully acknowledge the sponsorship of the Eurescom-funded CASPIAN project, P926.

References

1. ABONE Home Page. `www.isi.edu/abone/`.
2. D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network*, 1998. Special Issue on Active and Controllable Networks.
3. J. Ayars et al. Synchronized Multimedia Integration Language (SMIL) Boston Specification. Technical report, World Wide Web Consortium, June 2000. Working draft.
4. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.
5. R. Cardoe, J. Finney, A. C. Scott, and W. D. Shepherd. LARA: A Prototype System for Supporting High Performance Active Networking. In *IWAN*, 1999.
6. Component Compatibility Markup Language. `www.activenet.lancs.ac.uk/ccml/`.
7. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next-Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1), February 2000.
8. R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, IETF, 1997.
9. M. Fry and A. Ghosh. Application Level Active Networking. In *Computer Networks and ISDN Systems*, 1998.

10. A. Ghosh, M. Fry, and C. Crowcroft. An Architecture for Application Layer Routing. In *IWAN*, May 2000.
11. P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Computer Communications Review*, volume 29, pages 147–160, October 1999.
12. G. Hjálmtýsson. The Pronto Platform: A Flexible Toolkit for Programming Networks using a Commodity Operating System. In *OPENARCH*. IEEE, 2000.
13. Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
14. S. McCanne and V. Jacobson. The BSD Packet Filter: a New Architecture for User-level Packet Capture. In *USENIX*, 1993.
15. P. McKee and I. Marshall. Behavioural Specification Using XML. In *FTDCS*, Cape Town, December 1999.
16. S. Schmid, J. Finney, A.C. Scott, and W.D. Shepherd. Component-based Active Networks for Mobile Multimedia Systems. In *Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'2000)*, Chapel Hill, North Carolina, June 2000.
17. D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Operating Systems Review*, volume 34(5), pages 64–79, December 1999.
18. D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: Network Services Without the Red Tape. *IEE*, April 1999.