Dynamic Protocol Stack Composition: Protocol Independent Addressing

Sam Michiels, Tom Mahieu, Frank Matthijs, Pierre Verbaeten
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven
{Sam.Michiels,Tom.Mahieu}@cs.kuleuven.ac.be

June 2001

Abstract

This paper presents a generic addressing framework (GAF) which enables the development of protocol stack independent applications. This framework has been developed in the context of dynamic protocol stack composition [8]. Having a way to compose [8][7] and build [3] protocol stacks is not sufficient. The protocol layers a stack is composed of have an impact on the addressing used inside the stack. Since addresses are used by applications, the impact of modifying the stack dynamically is not automatically transparent. The addressing framework we present in this paper provides a generalised view on network communication to solve this addressing problem. We validated the GAF in a proof-of-concept prototype.

1 Introduction

This paper presents a generic addressing framework (GAF) which enables dynamic protocol stack composition. Dynamic stack composition comprises the building of a protocol stack which provides the required functionality to support an application. Applications that have totally different communication requirements can result in different protocol layer combinations. The goal is to allow applications to offer network services (such as irc, ftp, telnet), independent from the underlying network technology (e.g. IPv4, IPv6, ATM,...).

We have developed a protocol stack composition tool [8] that converts high-level application requirements into a suitable network protocol combination. Protocol layers provide and require specific functionality (such as transport, reliability or networking). This functionality is interpreted by the composition tool and matched with application-specific requirements to compose a protocol stack. The layers are developed using the DiPS (DistriNet Protocol Stack) component framework [3]. This framework supports the development of protocol stacks using a component-connector approach: functional DiPS components are glued together independently, using DiPS connectors that are modelled as first class entities. The approach of creating a path of connected components is similar to [1, 2, 9, 4]. The DiPS framework offers fine-grain components and large grain components such as protocol layers. Non-functional properties of a protocol stack, such as concurrency and the communication model are implemented in the connectors. The use of explicit connectors allows to adjust the flow at run-time: every connector can influence the packet flow; components can be replaced, added or removed since they do not have direct references to each other.

Being able to compose a protocol stack that adheres to application-specific requirements is not sufficient. The protocols in a stack have an impact on the addressing of applications that are using this stack. Since addresses are used by applications, stack modifications are not automatically

This research was supported by the Flemish Institute for the advancement of scientific-technological research in the industry (IWT) #990219(Pepita)

transparent. The addressing framework we present in this paper offers the necessary support to solve this problem.

Networked applications are developed with a specific network technology in mind. It is, for example, impossible for most applications to switch from TCP to UDP when network conditions would allow (or require) it (e.g. a telnet session on a local LAN without packet loss). Another example is the problematic conversion from IPv4 to IPv6 which requires all IPv4 applications to be re-engineered. The addressing solution is generic in the sense that our framework allows an application to use a different protocol stack depending on the environment it is deployed in.

In the following sections we describe how we solve the addressing problem in the context of dynamic stack composition. Section 2 discusses the most important abstractions such as generic socket, layer address and stack address. Section 3 handles address management at the layer and the protocol stack level. Section 4 presents the current realisations. We compare our work with other research projects in section 5 before we end with conclusions and future work in section 6.

2 Basic GAF abstractions

This section describes the GAF design. As seen in figure 1, GAF is an independent add-on for the existing DiPS framework, i.e. non-GAF-compliant protocol stacks can still be developed since the basic DiPS abstractions are not altered by GAF. In this section we discuss the core abstractions: the protocol independent application end point (`GenericSocket`) and support for addressing at the protocol stack level (`StackAddress`) and at the protocol layer level (`LayerAddress`).

2.1 Generic sockets

The application end point of a protocol stack is typically represented by a socket. Currently, sockets are protocol stack dependent. In the case of dynamically composed protocol stacks, an application that wants to communicate with a given service cannot know the socket-type to be used beforehand. The underlying protocol stack may for instance be an TCP stack or a UDP stack.

The GAF solves this socket-type dependency problem by introducing a generic socket. The `GenericSocket` class (figure 1) offers a protocol independent view on a socket. It offers functionality to create sockets, bind sockets on a local address, create sessions, send and receive messages or close sessions. A session is usually seen as the combination of a source and a destination stack address.

The notion of a separate session abstraction is intuitive, even in combination with UDP where no physical session is created. A user with an open (telnet, ssh, ...) session might pick up her laptop and disconnect from the network. After some period of time she reconnects at some other network location and expects that her session continues where it left off [6].

2.2 Addressing

Related to generic sockets is the issue of addressing. Current applications are protocol stack dependent and know exactly how to address a remote application end point. A GAF application that uses a generic socket does not know how to address an application end point since it is unaware of the protocols in the underlying stack.

We handle this addressing issue on both the layer level and the stack level. We introduce the concepts of layer addresses and stack addresses in the following sections.

LayerAddress

A layer address (`LayerAddress` class in figure 1) is a collection of one or more *primitive* addresses of a specific type such as IP. For instance, a layer address for UDP will contain a port number. A layer address for IP can contain a number of IP addresses on a multi-homed host. Figure 1 depicts this relation between the GAF `LayerAddress` and the traditional *primitive* protocol stack address.

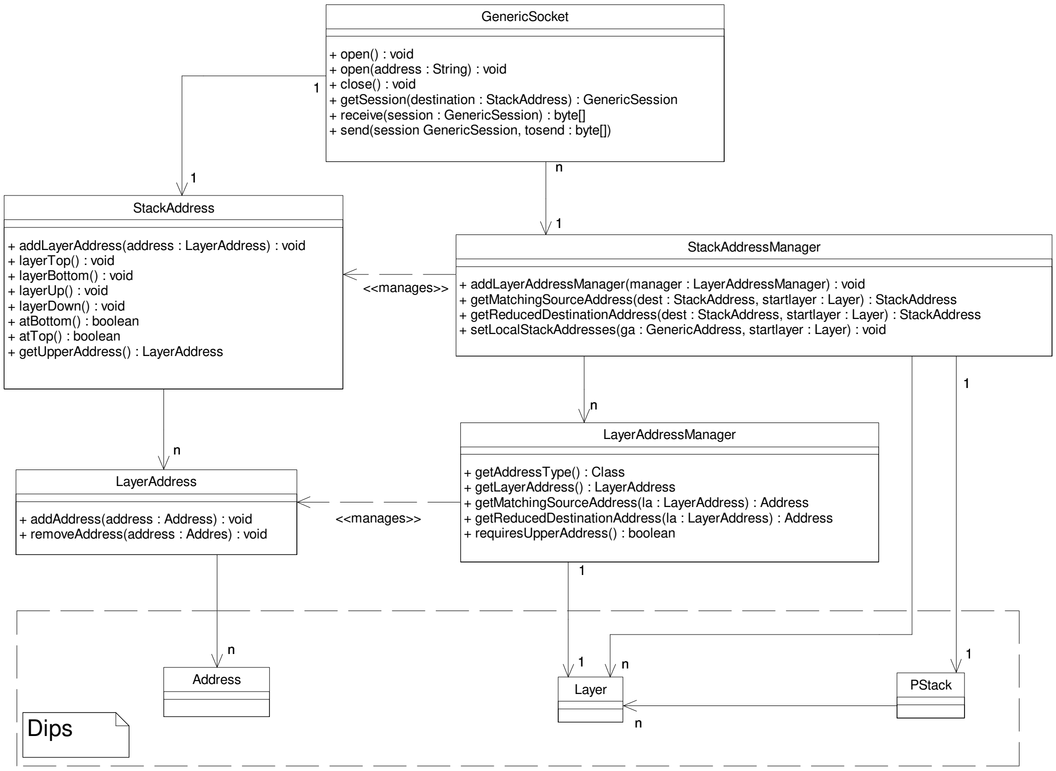

Figure 1: Generic Address Framework design

An IP layer address for a multi-homed host is textually represented as follows:

```
dips.protocol.ip.ipv4.IPv4Address/10.32.41.2/10.32.42.2
```

A layer address contains the type of the primitive address, which is a Java class name, and the actual address(es).

StackAddress

A stack address is a collection of layer addresses; one layer address for every layer in the stack. Textually, a stack address for a UDP/IP/Ethernet stack is represented as follows:

```
dips.protocol.udp.UDPAddress/8888|
dips.protocol.ip.ipv4.IPv4Address/10.32.42.123|
dips.protocol.ethernet.EthernetAddress/00:60:97:09:CA:C1
```

Applications use stack addresses to connect to services. Stack addresses are a way to define application end point without knowledge of the protocols (and by consequence the addresses) used in the underlying protocol stack. All needed protocol stack information is encapsulated in the stack address and hidden from the application programmer and end user.

Layers do not necessarily have to be address aware. For example, an encryption layer or a layer which provides customised reliability functionality does not require addressing. A dummy address is provided for such a layer, only to preserve a layered address architecture.

A stack address with a number of layer addresses possibly containing multiple primitive addresses may identify an application end point. It does not define a unique way to reach this application end point though. In case the IP layer contains multiple primitive IP addresses there may exist multiple routes to the service end point. Selecting the most appropriate primitive address for each layer in the stack is one of the main tasks of the layer address managers, which are discussed in the following section.

3 Address Management

Address management is needed since every `LayerAddress` contained in a `StackAddress` (see figure 1) must be reduced to a single primitive address in order to be useful for session initiation. The generic addressing framework offers two levels of address management: layer level and stack level address management. Layer address management is specific for each layer and is done by `LayerAddressManager`s (`LAM`s). Stack address management is done by a single `StackAddressManager` (`SAM`). Next to address reduction these managers also validate or provide a local layer or stack address. We will elaborate on these tasks for each level separately.

3.1 Layer level address management

Since every layer in a protocol stack has its own characteristics, address management tasks are delegated per layer to a specific `LAM`. A layer address manager knows how to deal with a primitive address type (such as UDP, IP or Ethernet). Similar to dummy layer addresses which we introduced for non-address-aware layers, a dummy `LAM` is transparently provided to preserve a layered address manager architecture.

A `LAM` is responsible for *matching* source and destination layer address, i.e. select a suitable destination layer address from a set of possible destination layer addresses given some source layer address, and the opposite; select a suitable source address given a suitable destination address. We first define the term *suitable* to make the discussion more focused. A `LayerAddress` is suitable when it contains only one primitive and valid address. A primitive layer address is valid when it defines a unique and possible path in the protocol stack in the context of a given stack address.

This matching process boils down to the following two tasks: on the one hand, a `LAM` provides a layer with a `LayerAddress`, i.e. it checks whether a given address is valid or it chooses a valid address if none is provided (`getLayerAddress()`). For example, the UDP address manager checks whether a given source layer address (i.e. a UDP port) is free and available for use. When no UDP layer address is provided the UDP `LAM` will provide one (i.e. allocates a free port).

On the other hand, a `LAM` is also responsible for reducing a given LayerAddress (`getReducedDestinationAddress()` and `getMatchingSourceAddress()`). Reducing a LayerAddress is the process of selecting an address from a given list of primitive addresses. This is for example what happens at the IP layer of a multi-homed host. A good criterion by which to reduce a destination layer address is to select an address from the source's network if possible. Traditional address translating techniques (such as DNS translation) often select one of the available IP addresses for a multi-homed host randomly since it lacks the additional source information.

Some address managers rely on a layer address from an upper layer to perform the matching process mentioned above. For example in a TCP/IP/Ethernet stack the IP layer will select the next hop address. This implicitly selects the local outgoing interface and the local Ethernet address too. Generally, a network protocol (such as IP) routes packets through the network and therefore influences addressing of lower layers (such as Ethernet). This upper layer addressing relationship depends on the composed stack. For example when no routing is needed, the IP layer can be skipped from the stack composition, resulting in a UDP/Ethernet stack. In this case the Ethernet address manager must be stand-alone (i.e. provide a valid match between source and destination Ethernet address). To handle this inter-layer dependent address management, we need a more global view on the protocol stack which is offered by the `SAM`.

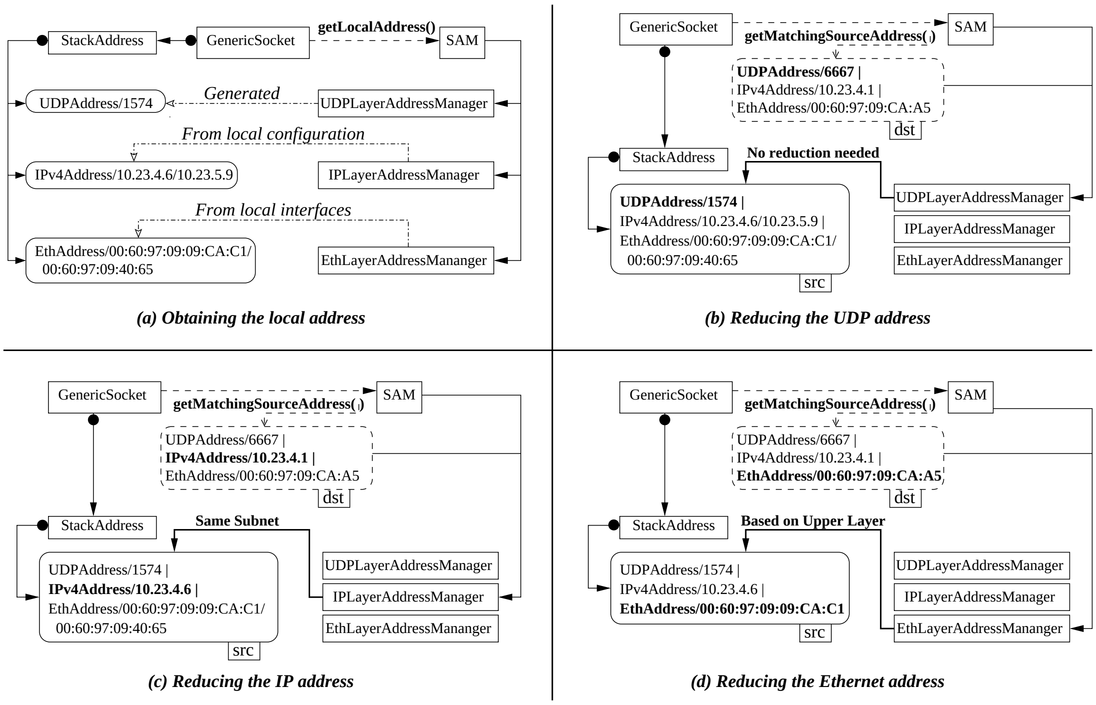

Figure 2: Address Reduction Example

3.2 Stack level address management

The task of the `StackAddressManager` (`SAM`) is to manage the protocol stack's addresses (source and destination stack address). This management is twofold: providing a local stack address for the composed protocol stack (`setLocalAddresses()`) and reducing the source and destination stack address (`getReducedDestinationAddress()` and `getMatchingSourceAddress()`). The latter is similar to the `LAM`'s task but at protocol stack level instead of at layer level. The `SAM`'s job is basically to travel through the protocol stack and cooperate with `LAM`s each managing the `LayerAddress` of one of the stack's layers.

The `SAM` is aware of the structure (i.e. the layer ordering) of the protocol stack. It manages the link between each layer and its `LAM`. During session initiation, when a stack address is being reduced, the `SAM` sequentially presents each `LAM` a `LayerAddress`. This can be a `LayerAddress` of the same type as that `LAM`'s address (for example in the UDP/Ethernet case) or it can be an upper layer `LayerAddress` in case the `LAM` requires an upper address (for example in the UDP/IP/Ethernet case) to make the correct reduction. The `LayerAddressManager.requiresUpperAddress()` and `StackAddress.getUpperAddress()` methods provide the `SAM` with the necessary information.

3.3 Example

Figure 2 depicts an example of address management for an application that wants to initiate an unreliable communication service with another application (in casu using a UDP/IP/Ethernet stack). The application communicates with a `GenericSocket` instance and is unaware of the actual protocol stack that is being used.

First the `GenericSocket` obtains a local address from the stack (figure 2a). The socket does this by contacting the `SAM` and asking it for a valid local stack address. The `SAM` contacts all `LAM`s to obtain the address(es) specific to that protocol layer. The `UDPLayerAddressManager` generates a free port number for the generic client socket that will be used by the application. The `IPLayerAddressManager` returns the IP addresses which the network interfaces are configured with. The `EthLayerAddressManager` returns the hardware addresses that are hard-coded on the Ethernet devices that appear to be present on the host machine. These addresses are wrapped in a `StackAddress` object and the `GenericSocket` is configured with the resulting stack address.

When the application wishes to communicate with another application, it needs to obtain a stack address of its peer[1]. The source and destination stack addresses need to be reduced for reasons explained earlier in this section. The reduction is done at session setup time. When the application requests a session with a given peer, the `GenericSocket` asks the `SAM` to reduce the local stack address. To do this, the `SAM` first contacts the `UDPLayerAddressManager` (figure 2b). The source UDP address already appears to be singular, the `UDPLayerAddressManager` does not have to do anything; the UDP address is already reduced. The `SAM` consults the next layer in the stack to further reduce the source address. The `IPv4LayerAddressManager` decides to select the source IP address that is on the same 2 as the IP address of the destination machine since that is obviously the shortest route (figure 2c). Subsequently, the `SAM` contacts the `EthLayerAddressManager` for reduction (figure 2d). The reduction criterion in the Ethernet layer is based on the reduction done in the layer above. The `SAM` knows about this upper IP layer and also about the reduced upper IP layer address. The `EthLayerAddressManager` consults the `SAM` to obtain this information, and uses it to select the Ethernet address that maps to the IP address selected by the `IPLayerAddressManager`.

The `SAM` now has a reduced source address. The same thing needs to be done for the destination address. In the case shown in the figure, this will be the same address as it already appears to be reduced (each layer address in the destination stack address is already singular). These two addresses can now be used to do the actual session initiation. Since we are working with a UDP stack, there will be no physical connection setup.

4 Implementation

We have a Java implementation of DiPS, our protocol stack component framework. Currently there are DiPS implementations of a.o. TCP, UDP, IPv4, IPv6, Ethernet. DiPS can be used as a user level process in combination with a slightly modified Linux kernel (only to allow direct communication with the Linux ethernet interface). We have also provided a special library that intercepts socket calls and delegates them to DiPS. This way, many Linux applications can transparently use DiPS sockets (without GAF support).

We have implemented GAF and combined it with DiPS and the composition algorithm. We replaced address aware DiPS components (a.o. header constructors/parsers) by GAF-compliant versions. All other components were reused. We were a.o. able to compose and run a UDP/IP/Ethernet and a UDP/Ethernet protocol stack without having to change any component or the application using the stack. Only a different EthernetAddressManager had to be used for each of the two versions.

5 Related work

Many component-oriented approaches to system software exist, such as 'Click' [2], 'Cactus' [9] and 'Scout' [4, 5]. These systems are comparable with DiPS, but they lack dynamic protocol stack composition and generic addressing as provided by GAF. Off++ [1] and Migrate [6] are systems that would benefit from or share common characteristics with GAF. These systems and their relation to GAF are discussed in more detail here.

In Off++ an OS kernel is developed that makes network resources (physical as well as system software resources, such as a TCP/IP stack) and automatic adaptation available to application and middleware programmers. Given the fact that network links between nodes can be quite different

[1] How this generic address is obtained is outside the scope of this document. A generic address can for instance be found by contacting a name server that distributes generic addresses instead of just IP addresses.

in nature (from ATM to wireless connections), the system has the same addressing problems as described in this paper. Resources can only migrate in the network if they can be addressed in a network protocol independent way.

In [6] a session-oriented end-to-end mobility architecture (`Migrate`) is described which supports sessions, independent from the underlying physical connection. The authors argue that a good mobile system architecture should a.o. have the following properties:

- eliminate the dependence of higher protocol layers upon lower-layer identifiers
- avoid prescribing a particular naming scheme
- handle unexpected network disconnections in a graceful way.

Protocol stack independent addressing is essential when mobile hosts are able to instantiate protocol stacks based on application requirements and available protocols. `Migrate` seems to focus on address migration (mobile hosts migrating to another network location), while we plan to support migration to a completely different protocol stack configuration in GAF (cfr. section 6).

6 Conclusions and Future Work

We have developed a framework that allows applications to use network services, independent from the protocol stack that is being used. This paper describes the consequences of stack independent addressing of network services and provides a solution for it: the generic addressing framework. This framework is used in combination with a stack composition tool and DiPS, a component framework which supports the development of protocol stacks. We have implemented and integrated GAF, DiPS and the composition tool in a proof-of-concept prototype.

An issue that still needs to be tackled is the address management of a stack with different protocols at the same level. An example of this is a protocol stack with IPv4 as well as IPv6 present. In this case support is needed to select the most appropriate address type instead of selecting a suitable primitive address. Providing several protocol stack instances for different combinations is not always a viable solution though: embedded systems for example have strict limitations on available resources. In this case a single protocol stack which enables several protocol combinations would be more appropriate.

The current GAF version supports composition of session-specific protocol stacks. We also want to support protocol stack adaptations *during* a session. Suppose a user with an open (telnet, ssh, ...) session using an IPv4 network connection; she might pick up her laptop and disconnect from the network. After some period of time she reconnects at the campus IPv6 LAN and expects that her session continues where it left off. We want the ability to dynamically add the IPv6 protocol to the stack, instead of throwing away the old stack and replacing it with a new one. The latter is not an option since existing IPv4 sessions would get lost. Keeping two stacks in the system until these sessions are closed may be impossible in embedded systems with minimal resources. The necessary support to integrate new layers in a stack at run-time is already provided in DiPS. However we still need support for dynamic stack and layer address management because we currently can not cope with address changes during a session.

References

[1] Francisco J Ballestros and Sergio Arevalo. The box: A replacement for files. *In Proceedings of HotOS-VII, AZ USA*, 1999.

[2] 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.

[3] Frank Matthijs. *Component Framework Technology for Protocol Stacks.* PhD thesis, Katholieke Universiteit Leuven, Dec 1999. Available at http://www.cs.kuleuven.ac.be/ samm/netwg/dips/index.html.

[4] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, Orcas Island, WA, May 1995.

[5] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 153–167, Berkeley, CA, USA, October 1996. USENIX.

[6] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering ip mobility. In *In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HoTOS-VIII)*, Schloss Elmau, Germany, May 2001.

[7] Ioana Sora and Frank Matthijs. Automatic composition of software systems from components with anonymous dependencies. *Submitted to 8th European Software Engineering Conference (ESEC)*, Mar 2001.

[8] Ioana Sora, Sam Michiels, Frank Matthijs, and Dirk Walravens. Policies for dynamic stack composition. Technical report, Dept. Computer Science, Katholieke Universiteit Leuven, Nov 2000.

[9] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *IEEE Infocom 2001, April 22–26, 2001. Anchorage, Alaska*, April 2001.