

Composing Active Services in NetScript

Position Paper, DARPA Active Networks Workshop, Tucson AZ, March 9-10, 1998

Sushil da Silva, Danilo Florissi, Yechiam Yemini

{dasilva, df, yemini}@cs.columbia.edu
<http://www.cs.columbia.edu/dcc>

Department of Computer Science
Columbia University
New York, NY 10027

March 1, 1998

Abstract

NetScript is a high-level programming language and environment for developing active network services and protocols. Programs written in NetScript can be deployed in nodes of an active network to dynamically extend their functions as needed. This paper presents an overview of NetScript and describes the three forms of composition that it supports: 1) composing active elements to form new protocols and services that are interoperable with existing protocols; 2) composing active elements to provide end-to-end services and functions; and 3) composing active elements to extend existing protocols with new capabilities.

1 Introduction

NetScript is a high-level programming language and environment for developing active networks services protocols. Programs written in NetScript can be deployed and installed in nodes of an active network to dynamically extend the network with new functions as needed. This paper provides an overview of the NetScript language focusing on its approach to composition of active elements.

Active networks give rise to several dimensions of composition. First is composition of active elements with their active node environment, particularly with other protocols executing at the active node. We use the term *composition for interoperability* to denote this type of composition. Traditional protocol architectures accomplish composition for interoperability primarily through layering and bridging. Second is composition of active elements with other elements to extend their functions. We use the term *composition for extensibility* to denote this type of composition. Third is composition of active elements in multiple nodes to provide end-to-end services. We use the term *end-to-end composition* to denote this type of composition. Traditional protocol architectures accomplish end-to-end composition by maintaining global flow relationships among nodes.

NetScript views an active element as an engine that operates on a stream of packets. It provides means to construct such active elements from primitive operators on packet streams; allocate node resources to support computations and communications by active elements; and control the flow of packet streams among active elements.

NetScript supports several forms of composition for interoperability. First, an active element can be layered above active or non-active components and use their services. For example, a new multimedia group communication protocol can be layered on top of an active real-time multicast protocol which, in turn, is layered over a variety of physical layer protocols as well as IPV4 and IPV6 protocols. Second, a NetScript active element can provide bridging as means to compose different active or non-active services. For example, one can use an active element to bridge two different multimedia/multicast services.

NetScript supports composition for extensibility by allowing one to extend an existing protocol with new functions and packet formats. For example, one can extend the POP3 mail protocol to provide authentication and access pro-

tection functions not available in the original protocol. Similarly, a NetScript active element can be used to extend a given protocol entity with instrumentation to monitor and analyzed intrusion attacks.

NetScript supports end-to-end composition by allowing active elements in multiple nodes to be interconnected to form Virtual Active Networks (VAN) providing end-to-end services. For example, a realtime multicast protocol could be deployed onto an entire VAN. This protocol would use a VAN's underlying virtual links to maintain multicast trees, and map tree link bandwidth to available resources. Furthermore, VAN mechanisms would enable security and protection of this protocol as a single end-to-end unit.

In what follows we review the NetScript language and focus primarily on its support of the three forms of composition discussed above.

2 NetScript Overview

This section provides a high-level overview of the NetScript language and its compositional model.

2.1 NetScript's Dataflow Computational Model

The power of NetScript as a language for composing active protocols comes from its underlying dataflow programming model. In a dataflow model, a computation consists of a collection of interconnected programs that process data streams flowing between them. Dataflow programs execute concurrently and their processing is reactive to the specific data arriving through data streams. NetScript views the task of an active node as that of processing packet streams and allocating resources to support this processing. Accordingly, the language supports construction of packet-stream processing engines, interconnecting these engines and allocating resources to them. NetScript programs communicate with each other through their input and output *ports*. The arrival of data at one or more input ports of a program triggers computation within that program.

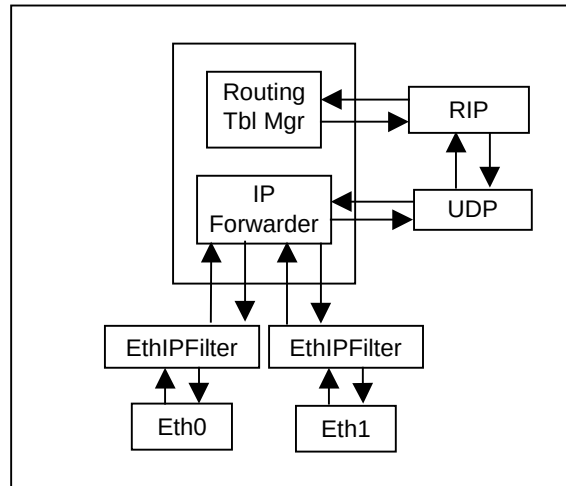


Figure 1: NetScript IP Router Configuration

Protocol composition is achieved by connecting together boxes. Thus, NetScript stream processing engines are built by connecting together or composing dataflow boxes to form a dataflow graph. The nodes in the dataflow graph are called *boxes* in NetScript in part because it helps to visualize a NetScript computation as collection of interconnected boxes. Boxes are constructed by connecting box output ports to input ports, subject to strong-typing constraints of the port types. Figure 1 illustrates the concept. The figure shows part of an IP router engine that we have implemented in NetScript. The boxes in the figure represent NetScript programs; the arrows are streams that connect output ports to input ports. A single output port can feed multiple input ports; similarly, multiple input ports can connect

to a single input port. Notice that a NetScript program can itself be recursively composed out of a network of inter-connected programs.

Dataflow is a good underlying model for NetScript because the model nicely captures the reactive nature of protocol processing—typically, the asynchronous arrival of messages from the network or from user processes triggers processing of protocol code. Another advantage of dataflow is that this model is already used to conceptualize and implement existing protocols (e.g. UNIX System V STREAMS).

2.2 Dynamic Composition of Protocols

What makes NetScript suitable for programming protocols in an active network is its support of dynamic changes in the dataflow graph; boxes can be added or removed, connected or disconnected at runtime. This property of NetScript enables dynamic composition of active protocols on a NetScript network.

NetScript provides a simple construct for dynamic composition of active protocols. This construct, called a *box template*, describes the input/output port signature and internal topological structure of a NetScript dataflow program. The port signatures define the types of messages a box will accept and generate. Its topological structure specifies interconnections between internal boxes. One can create one or more executable *instances* of a box template on active engines across the network. These box instances process packet streams that flow through them.

One deploys a new protocol on a NetScript network by simply defining a box template and dispatching it to individual NetScript engines in the network. NetScript uses Virtual Active Network (VAN)[1] mechanisms to enable macroscopic deployment and configuration of active protocols across entire networks. However, NetScript is *not* dependent on any particular deployment mechanism. In fact, NetScript templates can be embedded in packet-capsules and loaded on demand at NetScript nodes as these traverse a network.

When a NetScript engine receives a new box template, it creates a running instance of it by examining its port specification and internal topological structure. The engine then loads and instantiates each internal box, makes appropriate connections between box ports, and starts up the new instance to allow protocol messages to flow through it. Multiple box instantiations are overlaid upon underlying resources. Each box defines a protection boundary on the packets that flow through it. Some boxes reflect shared services (such as DNS); at most one (global) instance of a shared box template exists at any engine. When instantiating a template, NetScript automatically connects references to shared boxes to the single global instance of that box.

If a box template for any internal box does not yet exist locally, the engine downloads and installs it using a URL specified in the declaration of the enclosing template. This allows component parts of an active protocol to be developed by independent vendors and composed together to create value-added services.

3 Protocol Composition in NetScript

This section illustrates use of NetScript primitives to achieve three kinds of active service composition: composition for interoperability, composition for extensibility, and end-to-end composition. Due to space limitations of a white paper, we will only provide skeletal implementations that highlight compositional or language features.

Before proceeding, it is important to observe that NetScript is *not* a full-blown language for doing general-purpose computation. In particular, it does not include traditional control constructs (e.g., functions, iteration) and datatypes found in traditional sequential languages. Rather, it is helpful to view NetScript as *coordination* language [2] for gluing together *primitive* boxes to form arbitrary protocol processing engines. A primitive box consists of purely sequential event handling code that executes on arrival of messages at box input ports. Typical examples of primitive boxes include Ethernet device drivers, de-multiplexers, synchronizers, packet filters, message parsers, bounded buffers, routing table managers, queue policy managers, or full implementations of existing protocols such as IP.

Ordinarily, one builds *composite* NetScript boxes by composing primitive (or composite) boxes from a library. However, it is also possible to develop customized primitive components in a sequential language such as C or Java.

NetScript is implemented in Java and this facilitates easy construction of primitive boxes in Java through a simple API.

3.1 Composition for Interoperability

Composition for interoperability allows active protocols written in NetScript to interoperate with existing protocols through layering. The central idea here is that NetScript can be used to build either conventional standardized protocols (e.g., TCP/IP stacks), or to build new services that utilize a mix of standardized and active protocols. We show here the skeleton of an active real-time multicast protocol (RTMP) which is layered over IP. RTMPControl maintains multicast group membership and associated trees for RTMP. It uses underlying IP_RTMPMuxDemux to multiplex and demultiplex, parse and flatten RTMP PDUs that flow between RTMPControl and IP. The box template definition for RTMP appears below:

```
box template RTMP
{
  box import "http://cs.columbia.edu/ns/boxlib/RTMPControl.nbt" RTMPControl;
  box import "http://cs.columbia.edu/ns/boxlib/IP_RTMPMuxDemux.nbt" IP_RTMPMuxDemux;
  box import "http://ietf.org/ns/boxlib/IP.nbt" IP;

  connect
  {
    IP.rcvUp -> IP_RTMPMuxDemux.rcvDown,
    IP_RTMPMuxDemux.sndDown -> IP.sndUp,
    IP_RTMPMuxDemux.rcvUp -> RTMPControl.rcvDown,
    RTMPControl.sndDown -> IP_RTMPMuxDemux.sndUp,
  }
}
```

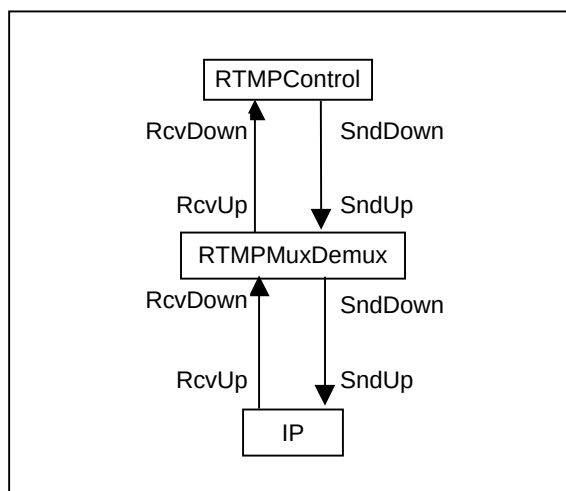


Figure 2: RTMP Protocol

This program defines a box template called RTMP. The connection topology of RTMP is shown in Figure 2. The first three lines of RTMP import internal box definitions for RTMPControl, IP_RTMPMuxDemux, and IP. The connect block defines the connection topology between these boxes. The '>' statement connects box ports together. When the RTMP box template is loaded at a NetScript engine, it uses the URLs of imported box templates to demand-load these templates as necessary and makes specified connections. If an imported box refers to a shared box template definition (e.g. IP) then connections are made to the single global instance of that box at the local engine. If IP were not yet installed at the local engine, NetScript would load it on demand and, in turn, resolve all of IP's dependencies (e.g. datalink devices).

3.2 Composition for Extensibility

Composition for extensibility allows one to extend an existing protocol with new functions and packet formats. For example, one can extend the POP3 mail protocol to provide authentication and access protection functions not avail-

able in the original protocol. To achieve this, one would define additional PDU structures for the new extended operations and then build a demultiplexer that splits the incoming stream and forwards PDUs to an appropriate boxes that handles each operation.

```

box template POP3
{
  inport popIn (in String cmd);
  outport popOut (out String reply);

  box import "popDemux.nbt" popDemux;
  box import "popState.nbt" popState;
  box import "popUser.nbt" user;
  box import "popPass.nbt" pass;
  box import "popRetr.nbt" retr;

  connect {
    popIn -> popDemux.in,
    popDemux.userCmd -> user.cmdIn,
    popDemux.passCmd -> pass.cmdIn,
    popDemux.retrCmd -> retr.cmdIn,

    user.reply -> popOut, user.setState -> popState.setState, user.getState -> popState.getState,
    pass.reply -> popOut, pass.setState -> popState.setState, pass.getState -> popState.getState,
    retr.reply -> popOut, retr.setState -> popState.setState, retr.getState -> popState.getState
  }
}

```

The program above is a box template definition for the extended POP3 protocol. It defines two ports, `popIn` and `popOut`. `popIn` is an input port (abbreviated `inport`) that accepts POP commands from a TCP connection; `popOut` is an output Port (abbreviated `outport`) that delivers replies to commands on the TCP connection. Incoming commands are demultiplexed by `popDemux` to boxes that handle each command type. Each command box (`user`, `pass`, `retr`) executes the incoming command and sends its reply on the `popOut` port. The command boxes also connect to a state box that maintains the POP state machine. Figure 2 shows the topology of the POP server. To simplify the diagram, the figure shows multiple copies of the `popState` (shaded). There is actually only one such box -- `user`, `pass`, and `retr` each connect to a single instance of `popState`.

New services can be added to the POP server dynamically. To do this, one would modify the box template for the POP service and deploy it across the network. NetScript engines would re-load the new template (i.e. replace the old version with the new) to support the extended services.

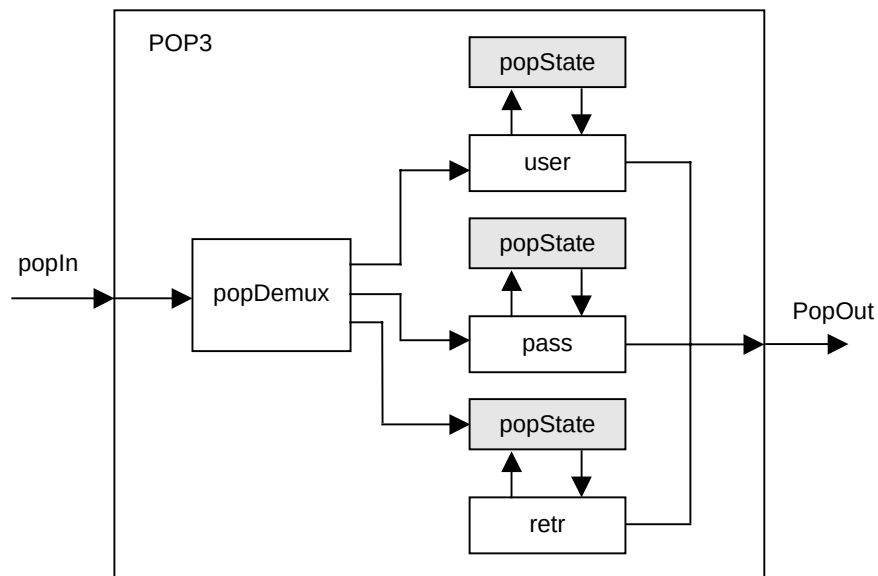


Figure 3: POP3 Protocol in NetScript

3.3 End-to-End Composition

End-to-end composition enables construction of active elements that when installed across an entire network performs some end-to-end function. Examples of such functions include maintaining a multicast group, per-application/per-network QOS management, or network monitoring. To accomplish end-to-end composition, NetScript uses a Virtual Active Network (or VAN) architecture to aggregate a collection of NetScript engines into a single virtual network for the purposes of code deployment, configuration management, security, protection, and end-to-end resource management. The central idea of a VAN is that instead of programming individual nodes independently, we treat a VAN as a single end-to-end unit.

A VAN is created by linking NetScript engines together through virtual links. The underlying transport semantics of a virtual link can be provided by a standard protocol (e.g. Ethernet, IP, TCP, ATM) or the link might utilize an active protocol written in NetScript. The particular choice reflects the needs of the VAN.

The VAN architecture is described in [1]. Here we show how a VAN is used to achieve end-to-end composition of active protocols. Consider for example the deployment on an active network of a realtime multicast protocol such as RTMP from the example above. To accomplish this, one would connect together appropriate NetScript engines across the network into a VAN and deploy RTMP directly on all nodes of this VAN. Because the VAN is the unit of security, authentication and access protection against unauthorized installation occurs just once and per-VAN when RTMP is installed. Because the VAN is the unit of protection, intrusions and failure are limited just to the compromised VAN. Because the VAN is the unit of resource management, RTMP can negotiate end-to-end resources such as bandwidth of virtual links on a per-VAN basis – RTMP maps and builds multicast trees based on link capacities of underlying links.

4 Conclusion

NetScript introduces mechanisms to construct and dynamically deploy protocols on active networks. It enables three forms of composition to support interoperability with existing protocols, to extend existing services standard or otherwise, or to introduce end-to-end services across a VAN. NetScript composition allows construction of new active services out of parts that include both active and traditional standardized protocols. In addition, NetScript can be used to construct active software at all protocol layers, from the datalink to the application.

5 References

1. da Silva, S., D. Florissi, and Y. Yemini. *Delegation-based Active Networks.. Submitted to SIGCOMM'98*. 1998.
2. Gelernter, D. and N. Carriero, *Coordination Languages and their Significance*. Communications of the ACM, 1992. **35**(2): p. 97-107.