Reasoning about Active Network Protocols

Ken Calvert

Department of Computer Science

University of Kentucky

Lexington, Kentucky

Samrat Bhattacharjee Ellen Zegura

Networking and Telecommunications Group

College of Computing, Georgia Tech.

Atlanta, Georgia

http://www.cc.gatech.edu/projects/canes

Sponsor: DARPA

Active Networking

Active networks provide a *programmable* user-network interface.

Users can:

- transmit packets
- *inject* code describing how [their] packets should be handled.

Benefits:

- Speed deployment of new services and algorithms.
- Improve service by exploiting the *combination* of application- and network-supplied information, e.g.: congestion onset; data dependencies.

Approaches to Programmability

Granularity

- per packet, in-band
- per flow, in-/out-of-band
- per node

What kind of abstract machine interprets the injected code?

- static (e.g. IP, ATM)
- pre-customized = code selects from menu (e.g. library)
- Turing machine

By Whom?

- end users vs. service providers vs. developers

The Problem

Conflicting Objectives:

- State global network properties that hold independent of injected code.
- Allow injected code to specify arbitrary behaviors.

Node behavior $\equiv$ fixed part + variable part

fixed part defines (e2e) behavior:	network properties easier to show
	limited flexibility
variable part defines (e2e) behavior:	full flexibility
	can't prove much *a priori*

CANEs Approach

- Define *generic* packet processing behavior(s) of nodes.
- Define specific points (slots) where behavior can be modified.
- Provide canned behaviors to go in slots, allow *injection* of user-defined slot programs.

Example: Forwarding Behavior

Parse packet, obtain *src, dest, fwding table id, auth token*

$\langle$**Slot 0:[null]**$\rangle$ $\{$marker to src, cache payload, send ack to prev. hop$\}$

i := **Lookup**(*src, dest, fwding table*)

if $i = \perp$ then $\langle$**Slot 1:[null]**$\rangle$ $\{$error message to *src*$\}$

$\langle$**Slot 2:[null]**$\rangle$ $\{$snd i to *src*, authenticate $i\}$

if i is congested then $\langle$**Slot 3:[discard]**$\rangle$ $\{$queue manipulation$\}$

$\langle$**Slot 4:[null]**$\rangle$ $\{$(local) smoothing, scheduling$\}$

enqueue packet for i.

Define services by injecting/selecting code in slots.

Language Independent Active Network Environment

Active node behavior defined by **underlying program**, plus **injected program(s)** bound to **slots**.

- A formal model using UNITY notation and logic
- Underlying programs interact with injected programs via shared variables.
- Slots are *raised* to enable the injected code.
- Each slot has resource bounds, restrictions and obligations of injected code. (Syntactically checkable.)

Why UNITY?

- Single composition operator $\|$ allows a simple model of injection and resource-bounding mechanisms.
- Well-understood logical machinery.

Underlying Program

Program {**Node**} *Program at each active node v*

initially

N0 $v.state, discCnt, errCnt = idle, 0, 0$ { Initialization }

assign

N1 $\langle \langle \, [\!] \, x : v.inC[x] \in v.inC :$
$v.state, v.inC[x], v.Msg, v.LH := newPkt, \mathrm{tail}(v.inC[x]), \mathrm{head}(v.inC[x]), x\rangle$
$\| \ \langle \| \, i :: v.rt.i.usage := 0\rangle$
$\rangle$ if $v.idle \wedge (v.inC[x] \neq \bot)$
{ If channel is non-empty, read message and initialize usage counters }

N2 $[\!]$ $v.state := slot.0.raise$ if $v.newPkt$ { Raise message arrival event }

N3 $[\!]$ $v.state, v.NH := rtFound, v.RtTable(v.Msg.d)$ if $v.slot.0.cmpl$
{ Route message to proper channel }

N4 $[\!]$ $v.state := slot.1.raise$ if $v.rtFound$ { Raise routing done event }

N5 $[\!]$ $\langle v.state, \; v.outC[v.NH] := idle, \; v.outC[v.NH]; v.Msg$
$\| \ \langle \; discCnt := discCnt + 1 \;$ if $\; end(v.outC[v.NH]) = NullProc$
$[\!] \; errCnt := errCnt + 1 \;$ if $\; end(v.outC[v.NH]) = ErrProc\rangle\rangle$
if $v.slot.1.cmpl$ { Send message on proper channel; Update Counters }

end $\{Node\}$

Underlying Program — Default Slot Behavior

Program $\{$**DS**$\}$ *Default Slot*

initially

D0 $\langle [\!] \, i :: v.rt.i.usage, v.rt.i.bnd = 0, \beta_i \rangle$ $\{$ Initialization, $\beta_i \geq 0$ $\}$

always

D1 $\langle [\!] \, i :: v.SlotCnd.i = (v.rt.i.bnd > v.rt.i.usage) \wedge v.slot.i.raise \rangle$

$\{$ Default set of conditions for progress through slot $\}$

D2 $\langle [\!] \, i :: v.Prog.i = Q.i \rangle$

$\{$ "background" predicate Q, set to $\mathtt{true}$ if no programs are bound to slot i $\}$

assign

D3 $\langle [\!] \, i :: v.rt.i.usage := v.rt.i.usage + 1$ if $v.SlotCnd.i \wedge v.Prog.i \rangle$

$\{$ Increase resource usage if no other program active $\}$

D4 $[\!] \, \langle [\!] \, i :: v.state := v.slot.i.cmpl$ if $v.slot.i.raise \wedge v.rt.i.bnd = v.rt.i.usage \rangle$

$\{$ Resource bound exhausted, slot processing complete $\}$

end $\{DS\}$

General Results

Definitions

- Well-formedness (*receptivity*) of underlying program
- Well-formedness (*acceptability*) of injected program
- Injection transformation, combines with default slot program

Metatheorems

- Injection preserves receptivity.
- Injection distributes over $\|$.
- Injection preserves properties of (underlying program $\|$ injected program).
- Injection preserves *pure* properties of injected program, modulo resource bounds.

Properties of Underlying Program

- Messages eventually reach their destinations.

Example: Mobility

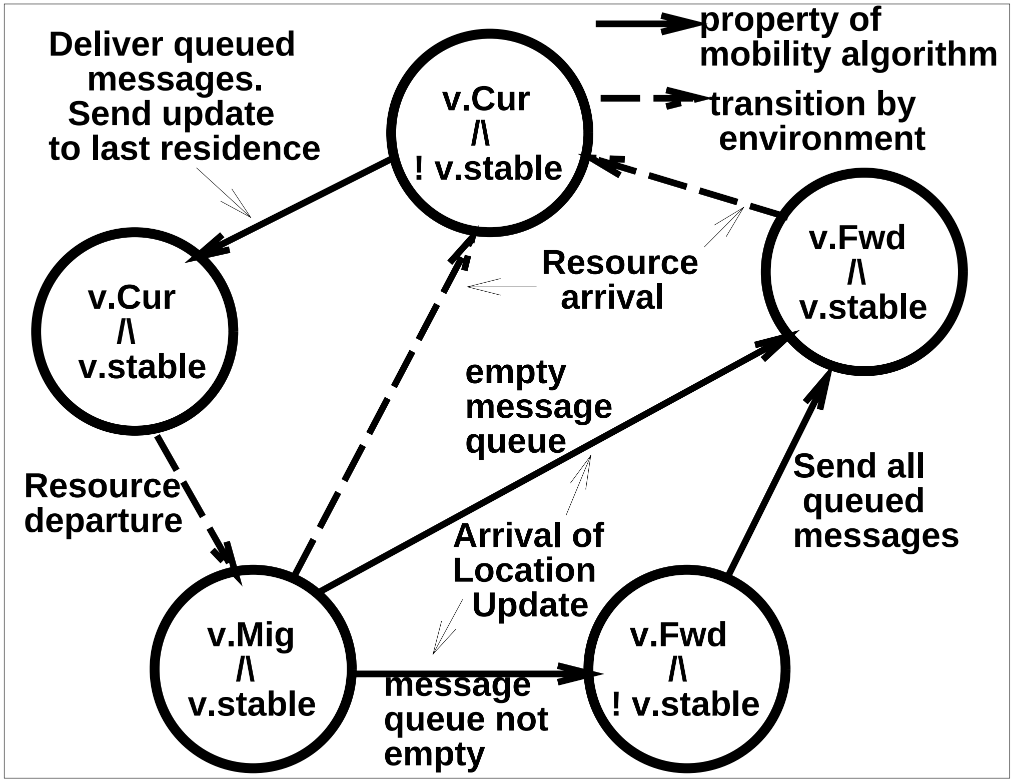
The Problem

- A *resource* migrates spontaneously from node to node.
- Messages are addressed to the “last known address” of the resource.
- Nodes keep pointers to resource location, forward messages toward it.

The Approach

- Bind code for mobility to slot 0.
- Messages for the resource carry last-known location, plus a (logical) timestamp.
- When nodes see messages with newer timestamps, they update their pointers to the resource.
- When resource arrives at a node, it increments timestamp and sends an update to the previous location in the message.

Mobility Example



Properties

- Messages not addressed to the resource reach their destination.
- Messages reach either the resource or a node with newer information.

Mobility Example

Program $\{$**Mobility**$\}$ *Mobility Code for Slot 0*

initially

MA0 $v.rState, v.rLC, v.rLoc, v.rStable, v.rQ = Fwd, 0, r.home, \text{true}, \perp$
$\text{if } v \neq r.home \sim Cur, 0, v, \text{true}, \perp \text{ if } v = r.home$

$\{$ Resource r is initially located at $r.home$; this is known to all other nodes $\}$

assign

MA1 $v.Msg.d, v.Msg.loc, v.Msg.ts := redir(v.rLoc, v.Msg.d), v.rLoc, v.rLC$
$\text{if } v.rLC > v.Msg.ts \wedge v.Msg.type = Access \wedge (v.Fwd \vee v.Cur) \wedge v.stable$

$\{$ Re-direct accesses containing stale information $\}$

MA2 $v.rLoc, v.rLC := v.Msg.loc, v.Msg.ts \text{ if } v.rLC < v.Msg.ts \wedge v.Fwd \wedge v.stable$

$\{$ Update local clock and forwarding information if message contains newer information $\}$

MA3 $\langle fwd\ (Qh.s, redir(v, Qh.d), Qh.r, v, res.ts + 1, Qh.type, Qh.body)$
$\parallel\ v.rQ := \text{tail}(v.rQ)\rangle \text{ if } v.Cur \wedge \neg v.stable \wedge v.rQ \neq \perp$

$\{$ Resource arrives at node v; Deliver all queued messages $\}$

. . . etc.

Finis

Conclusions

- A model of active node programming using UNITY
- The slot model is intended to permit reasoning about global behavior with limited knowledge of the injected program.
- We still need strong/precise constraints on the injected program to guarantee underlying properties (e.g. every message reaches its destination).
- Mobility as an application for active nets

Future Work

- Other applications: reliable multicast. . .
- Reasoning about behavior *during* injection, when some nodes have the injected code and some don't.