Design and Implementation of Composite Protocols

Magesh Kannan

Master's Thesis Defense
The University of Kansas
12.16.2002

Committee:
Dr. Gary J. Minden (Chair)
Dr. Joseph B. Evans
Dr. Victor Frost

Outline

- Motivation
- Elements of a Protocol Component
- Composite Protocol (CP) Framework
- Implementation of CP Framework over Ensemble
- Performance Evaluation
- Summary & Future Work

Motivation for Composite Protocols

- Layering only a design principle
  - Implementations usually less-structured
- Correctness as important as efficiency
  - More so with active networks
  - Typical implementations only tested; not proven correct
- Code reuse not prevalent in protocol software
- Few choices in configuring a protocol
  - e.g. simple UDP or feature-rich TCP
- Developing variants not easy

Advantages of Composite Protocols

- Protocols implemented as collections of single-function components
  - Formal verification tasks more manageable
  - Better scope for reuse of protocol functions
  - Faster development of variants
  - Better prospects for customization
  - “Properties-in Protocol-out”

Definitions

- Protocol Component
  - Single-function entity
  - Not very useful stand-alone
  - e.g. fragmentation, reliable delivery, neighbor discovery
- Composite Protocol
  - Collection of components arranged in an orderly fashion
  - Useful as a unit
  - e.g. file transfer, web document retrieval, byte-stream transport
- Network Service
  - Collection of cooperating composite protocols that offer a larger communication service
  - e.g. multicast, web caching

Composition Method: Linear Stacking

- Composite Protocol -> Linear arrangement of components
- Messages from application processed by a fixed sequence of components
- Sequence decided at configuration time and does not change during operation
- Regularity of processing sequence helpful for formal reasoning
- Absence of multiplexing and demultiplexing
  - Each application gets its private instance of a composite protocol
- Need for stack matching
  - Identical sequence of components at endpoints and routers

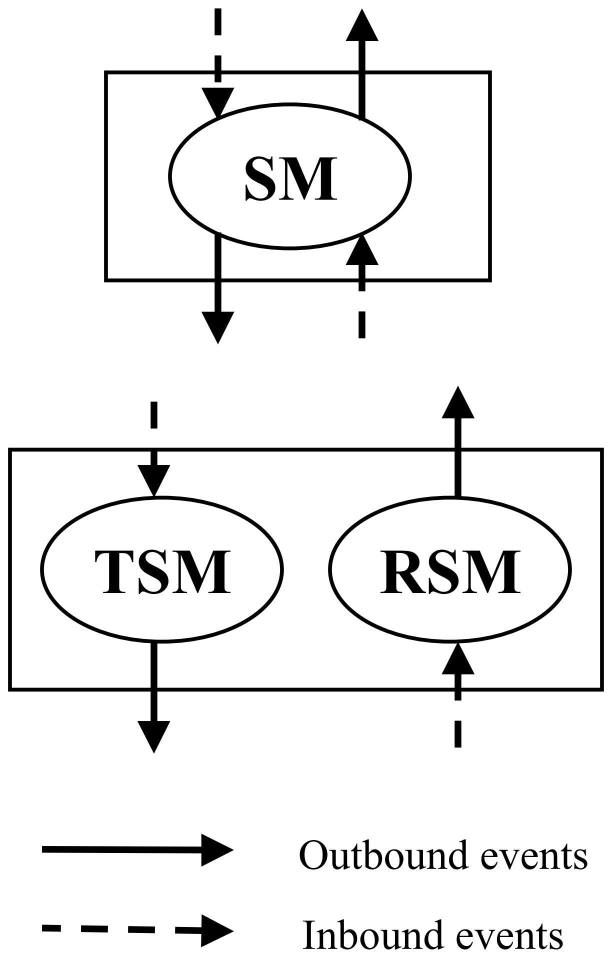
Elements of a Protocol Component

- State machine representation
- Memory model
- Formal properties
- Parameters
  - Knobs to tune component functionality
- Control interface

State Machine Representation

- Component functionality as an FSM
  - Natural representation for protocols
  - Facilitates automatic verification and validation
- Augmented state machine model
  - Guard expression
    - predicate that should hold TRUE for transition to be executable
  - Synchronous transition
    - “no-wait” transition
    - triggered upon entry to a state

Component as a State Machine

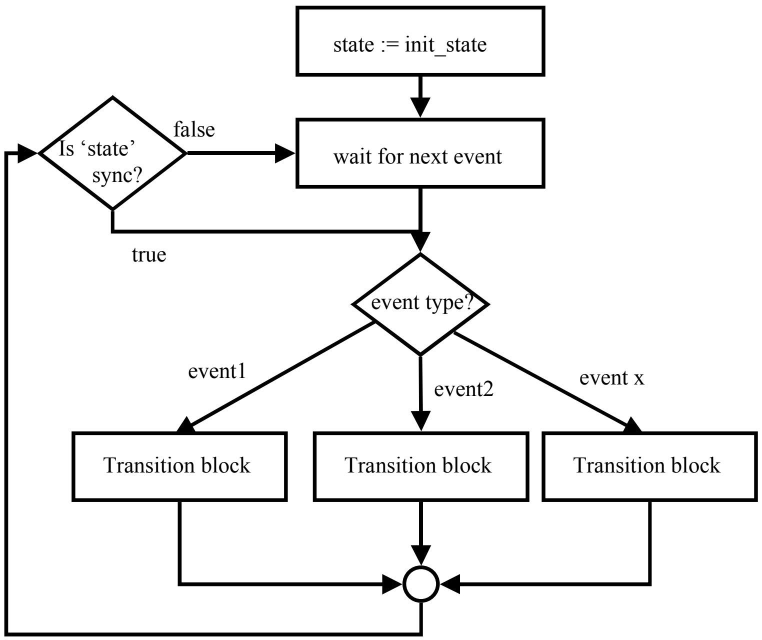


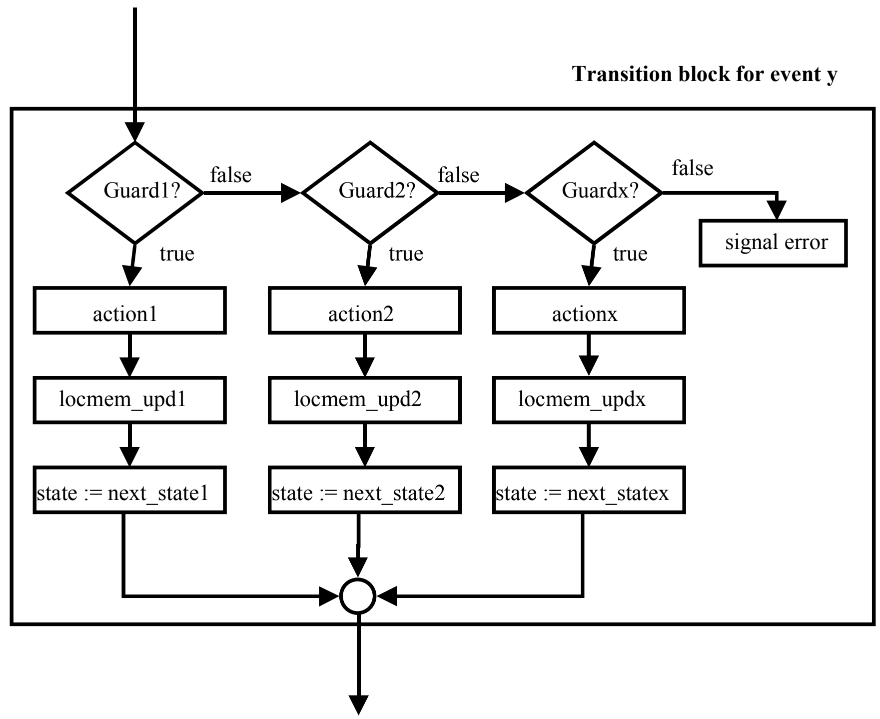
- Transitions are atomic
- Events to busy SM are queued
- Component
  - driven by two event queues
  - actions generate events that are deposited into one of two output queues
  - either one or a pair of state machines (transmit and receive)
  - pair of state machines if the transmit and receive tasks are relatively independent

State Machine Structure

- State Machine
  - List of States
- State
  - List of Transitions
- Transition
  - *current state*
  - *next state*
  - *event type*
  - *guard expression*: condition for this transition to be executable
  - *action:* response of the SM for this event
  - *local memory update*: changes to local memory objects

State Machine Execution

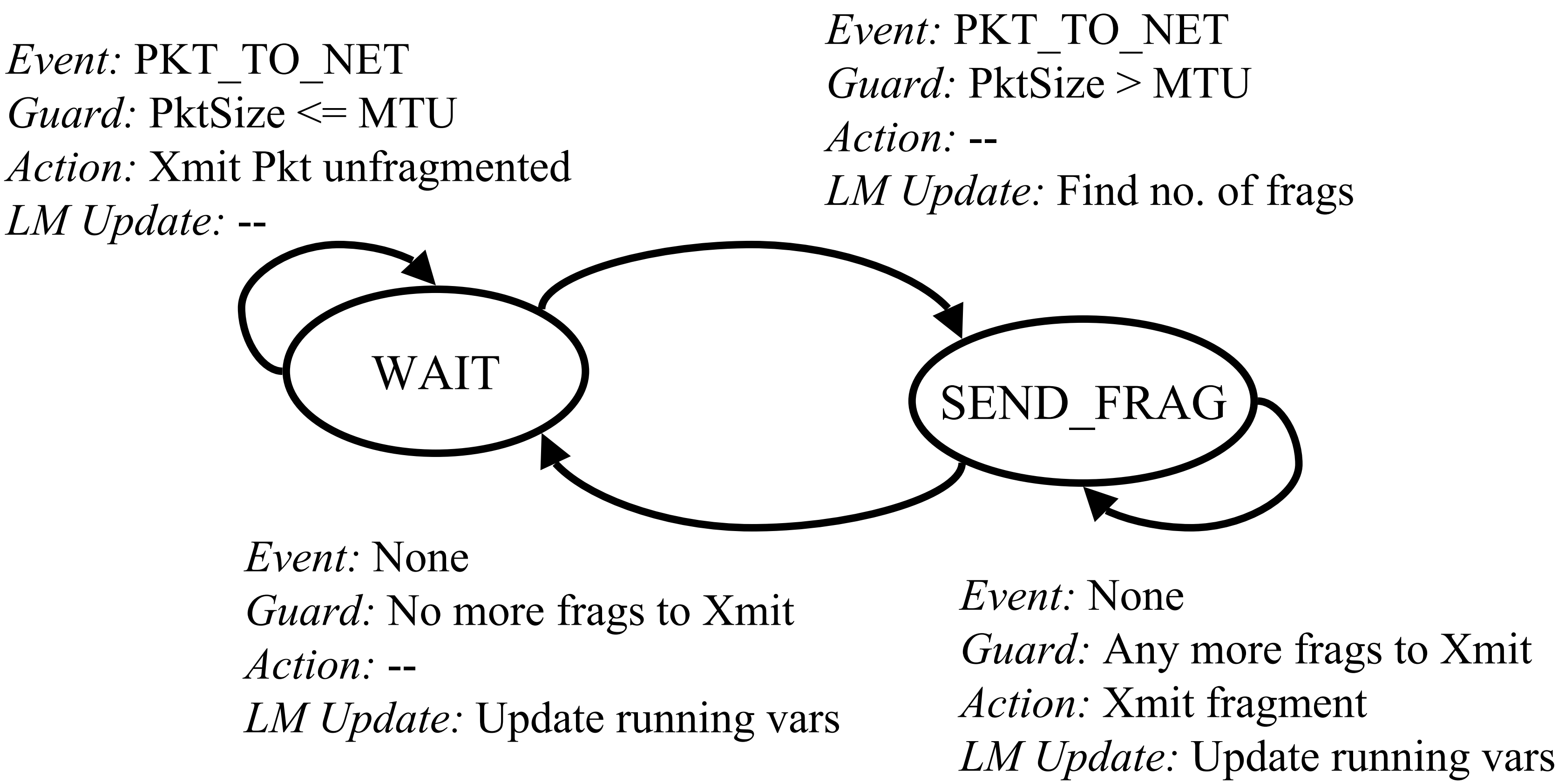




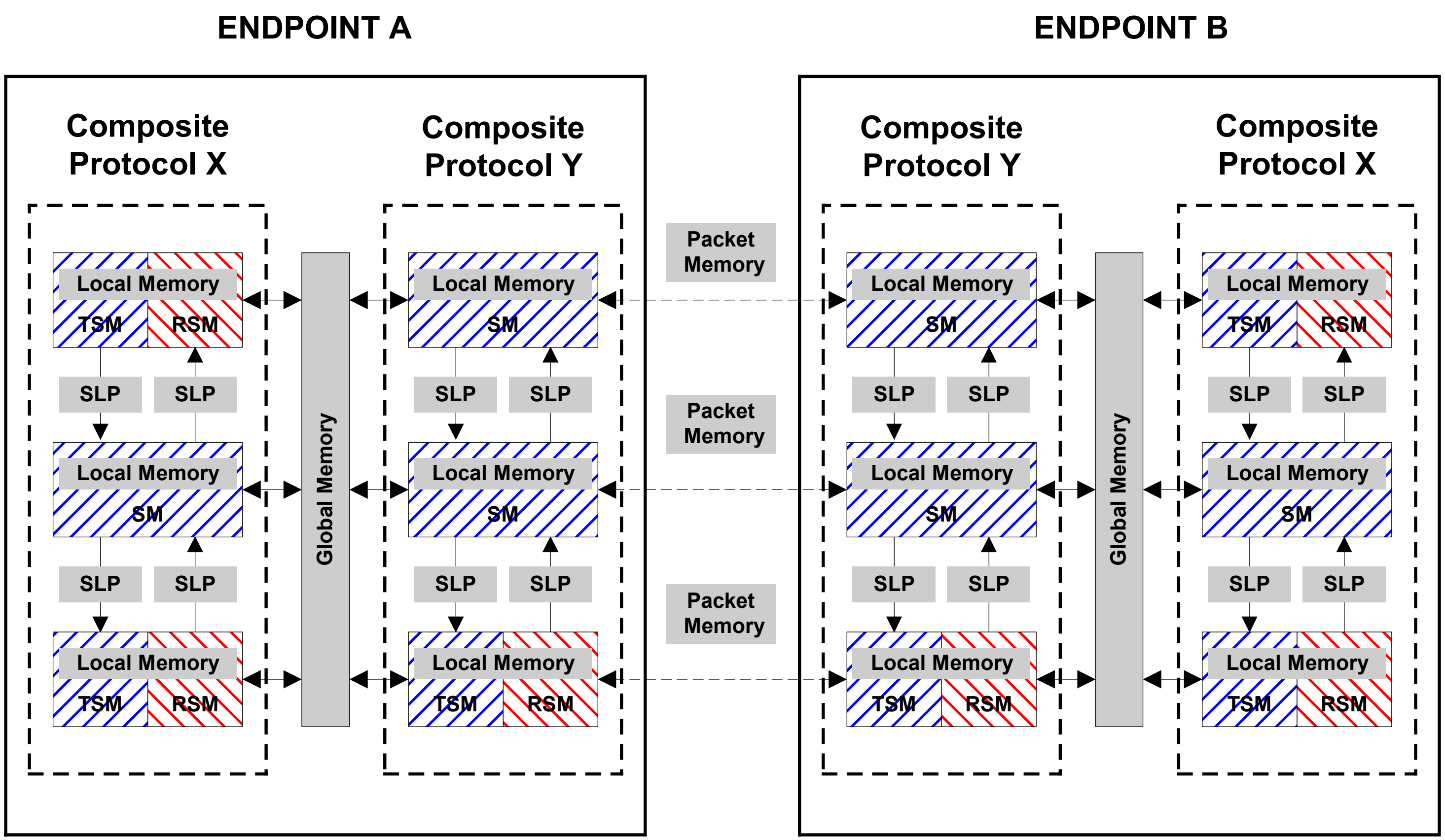
State Machine Semantics

- Guard Expressions
  - Only one of the guards shall be TRUE for any event occurrence
  - None of the guards is TRUE
    - under-specified state machine
  - More than one guard is TRUE
    - ambiguous state machine
  - Purely functional
    - no side-effects
- Synchronous Transitions
  - If one of the transitions from a state is synchronous, all transitions shall be synchronous
- Imperative behavior limited to local memory update function

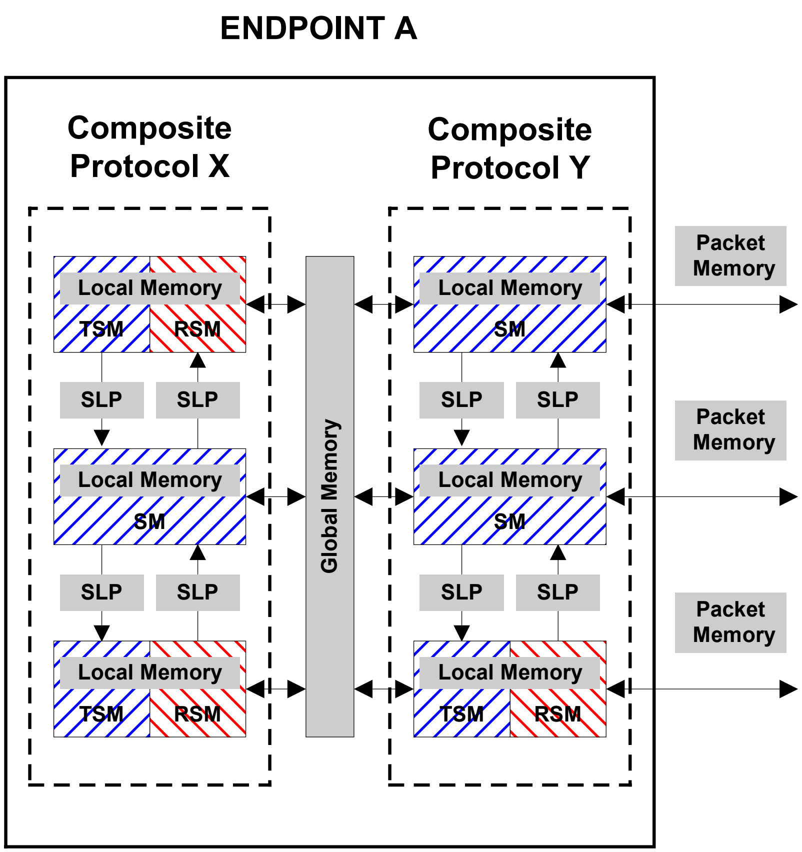
Example: Fragmentation TSM



Memory Model

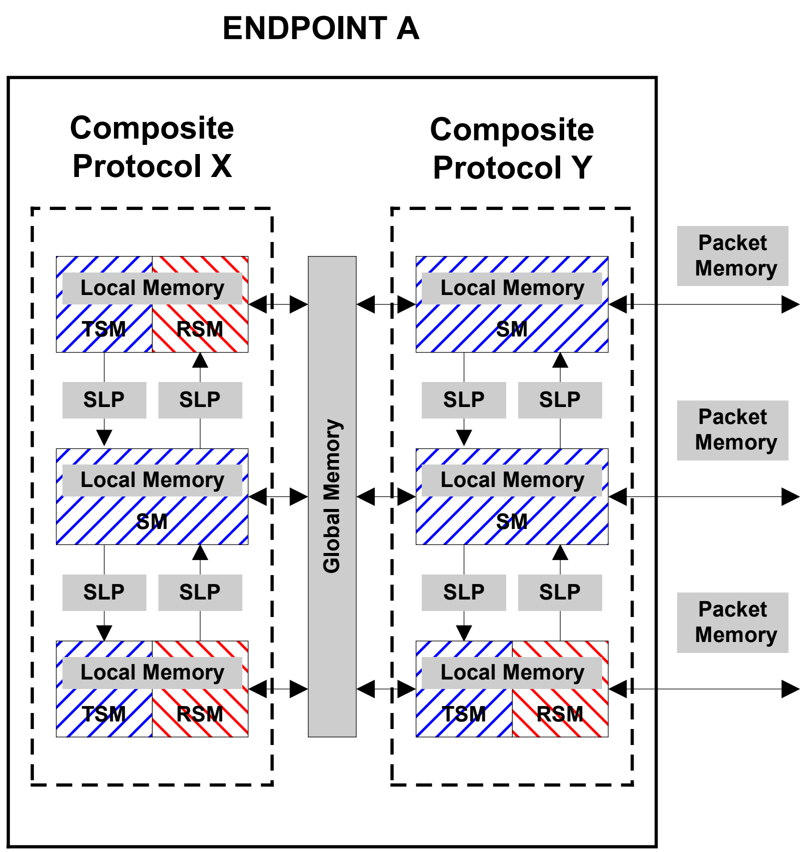


Memory Model: Packet Memory



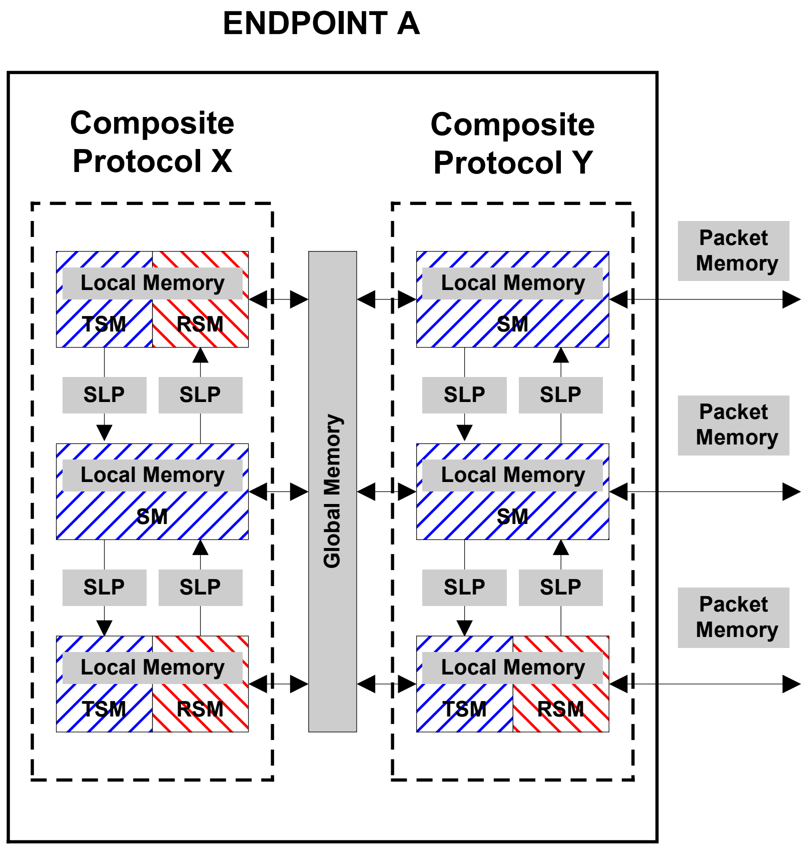
- Term for header fields attached to packets
- A kind of memory because it transfers state info. across peer components
- Accessible only to peer component instances
- Read-only transparent access to other lower-level components
- Extent same as transit time of packet between peer components
- e.g. checksum, sequence numbers, fragment identifiers

Memory Model: Local Memory



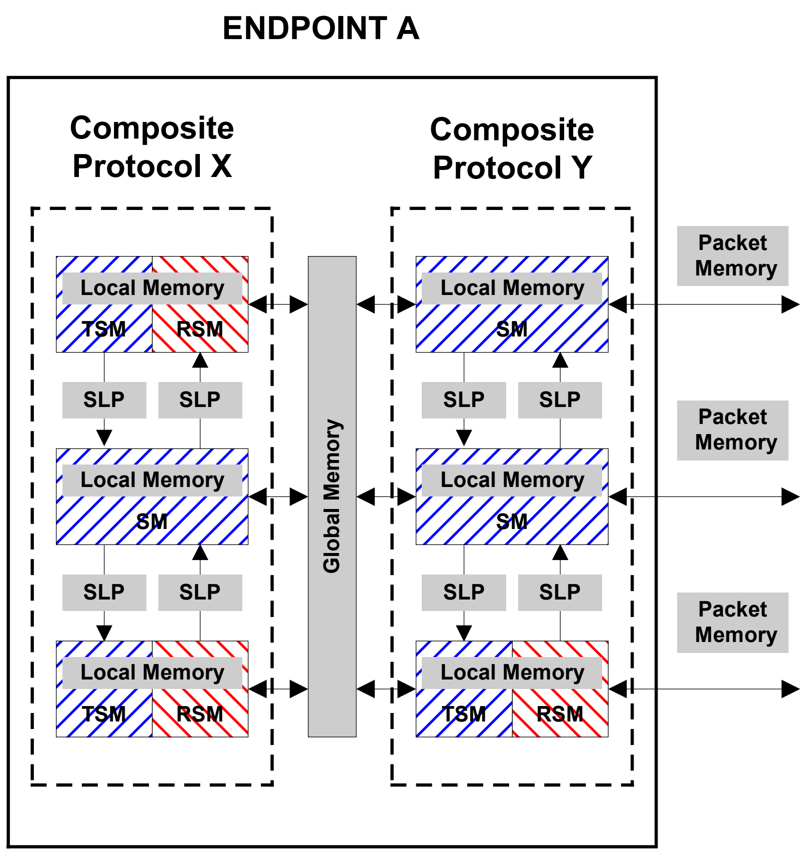
- Local to one instance of a component
- Extent same as that of component
- For TSM+RSM, accessible to both state machines
- Only one of TSM and RSM active at any time, hence no concurrent access
- e.g. unacknowledged packets, incomplete fragments, neighbor identities

Memory Model: Stack-local Packet Memory



- Pertains to a packet but is local to a composite protocol instance
- Accessible to all components in a composite protocol instance
- Extent same as that of the packet
- Strong requirements to avoid incompatibilities
- e.g. next hop network address, time-to-live

Memory Model: Global Memory



- Shared by more than one composite protocol at an endpoint
- Accessible to all components of all composite protocol instances at that endpoint
- Functional interface to manage concurrent access
- Extent same as that of endpoint OS
- e.g. routing table entries, multicast group membership info.

Formal Properties

- Assertions about a condition being TRUE for a packet or a sequence of packets
- e.g. bit-error free transmission, in-order delivery
- Requirement
  - a property that shall hold for correct operation of a component
  - e.g. no bit errors in packet memory fields
- Guarantee
  - a property provided by a component, given the requirements
  - e.g. in-order delivery of packets
- Invariant
  - a property preserved by a component
  - e.g. encryption component preserves in-order delivery

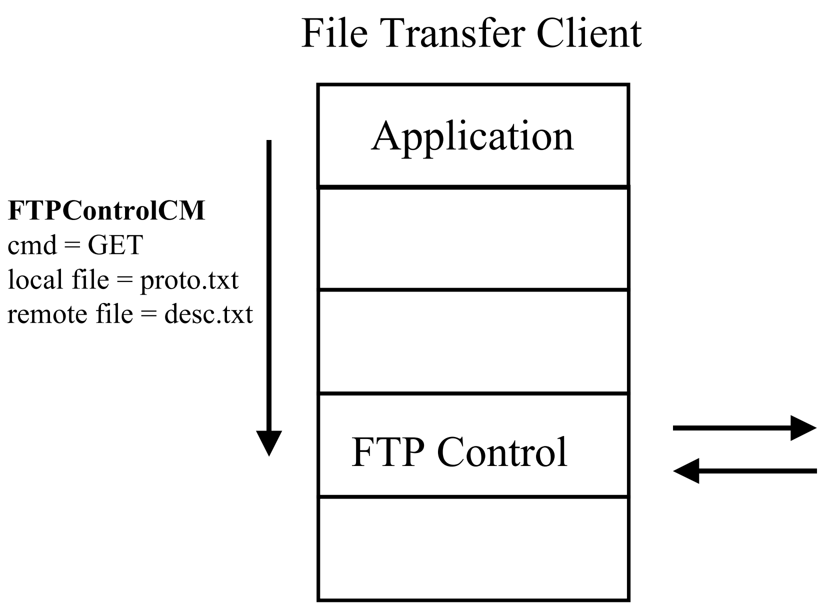
Control Interface

- Mechanism for inter-component and intra-composite protocol communication
- Leads to development of smaller components that are likely to be reused more often
- e.g. file transfer as FTP control and FTP data components
  - FTP control and FTP data
    - web caching
    - file transfer
  - FTP data alone
    - streaming multimedia
    - data logging

Design of Control Interface

- Modeled as exchange of messages between components
- Controlled component offers a service
- Controlling component uses the service
- Service
  - Specification of service request
    - unique name
    - list of commands & parameters for each command
  - Implementation
    - transitions for every command of the service in the SM of controlled component, one command at a time
    - commands carried by CONTROL events
  - Invocation
    - creation of a service request and a CONTROL event to carry it

Control Interface Example



- Service specification
  - name: FTPControlCM
  - *commands* (parameters)
    - *user* (name, pw),
    - *get* (local, remote),
    - *put* (local, remote),
    - *list* (remote),
    - *quit*
- Service Implementation
  - peer-to-peer communication initiated upon reception of service request

Composite Protocol Framework

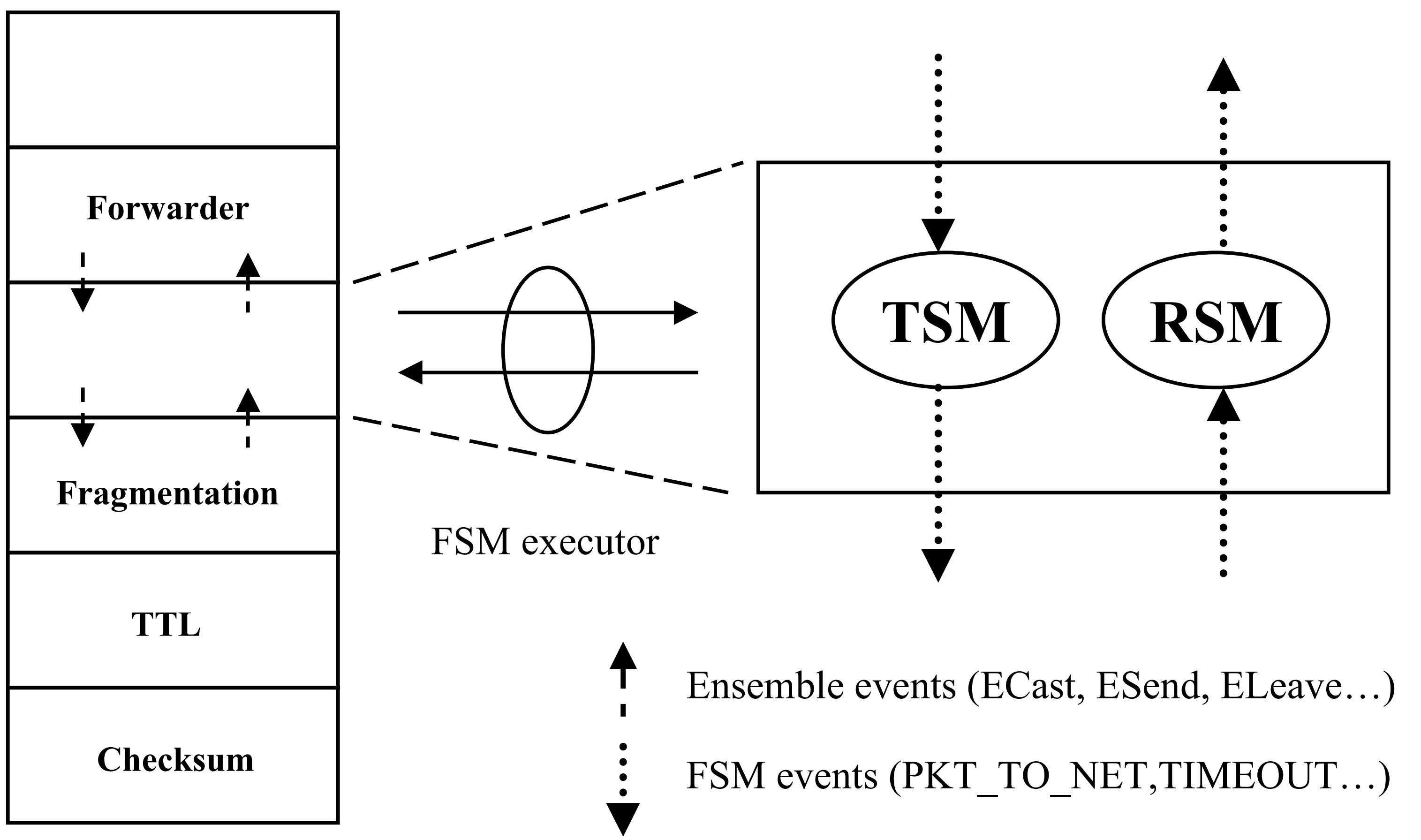
- Infrastructure for composition and operation of composite protocols
- Responsibilities
  - Drive state machines of components
  - Manage event queues between components
  - Map packet receptions to state machine events and vice versa
  - Implement primitives
    - e.g. PktSend, PktDeliver, NewPktSend, NewPktDeliver etc.
  - Provide an application interface

Overview of Ensemble

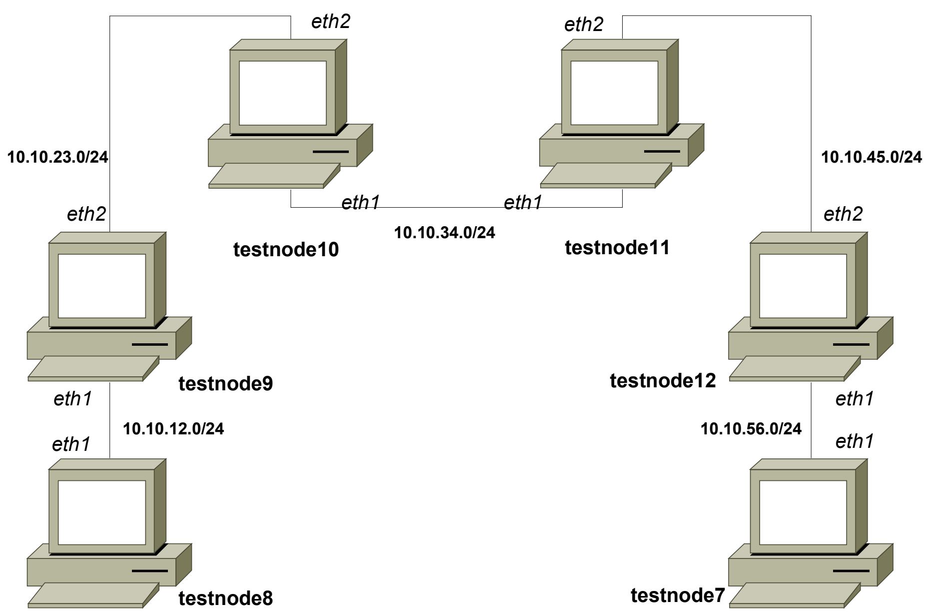
- Group communication system developed at Cornell
- Unit of composition: Layer
- Supports linear stacking of layers
- Event handlers executed atomically
- Implements unbounded event queues between layers
- Implemented in OCaml
  - Better prospects for formal analysis
  - Prior reported results on analysis using NuPrl
- Layers offer a uniform interface

FSM executor over Ensemble

Custom Composite Protocol



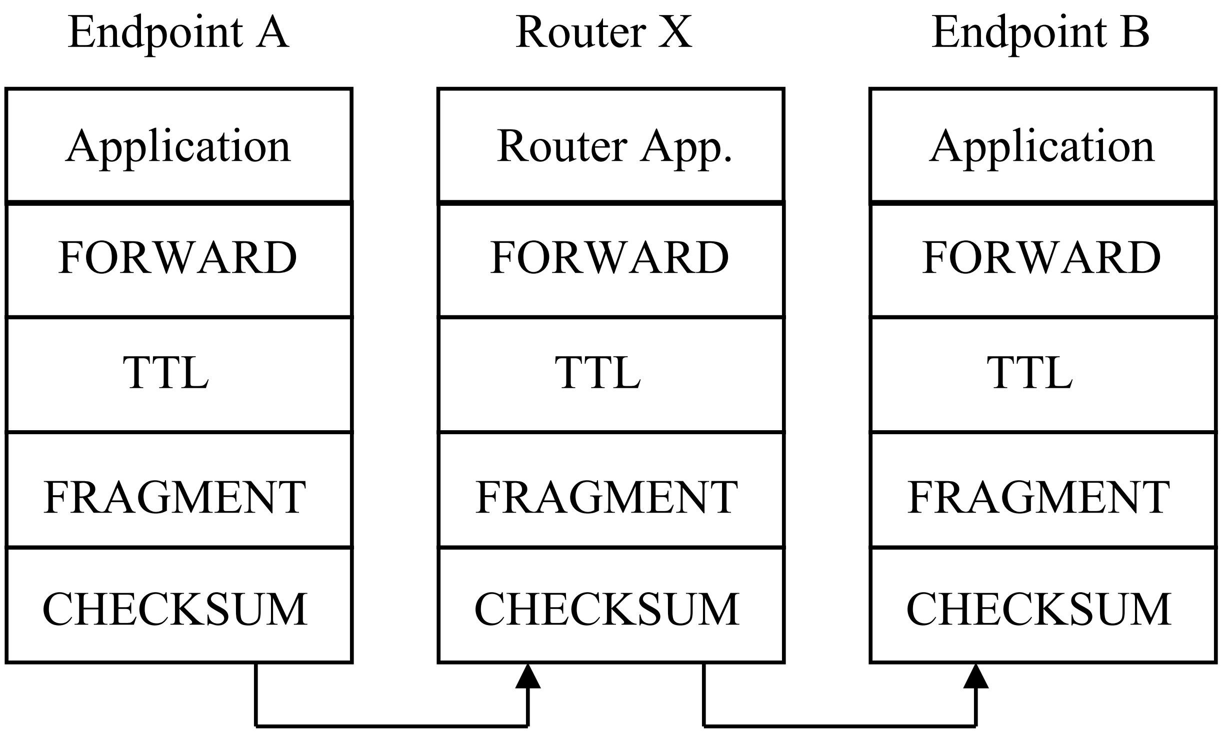
Performance Evaluation: Test Setup



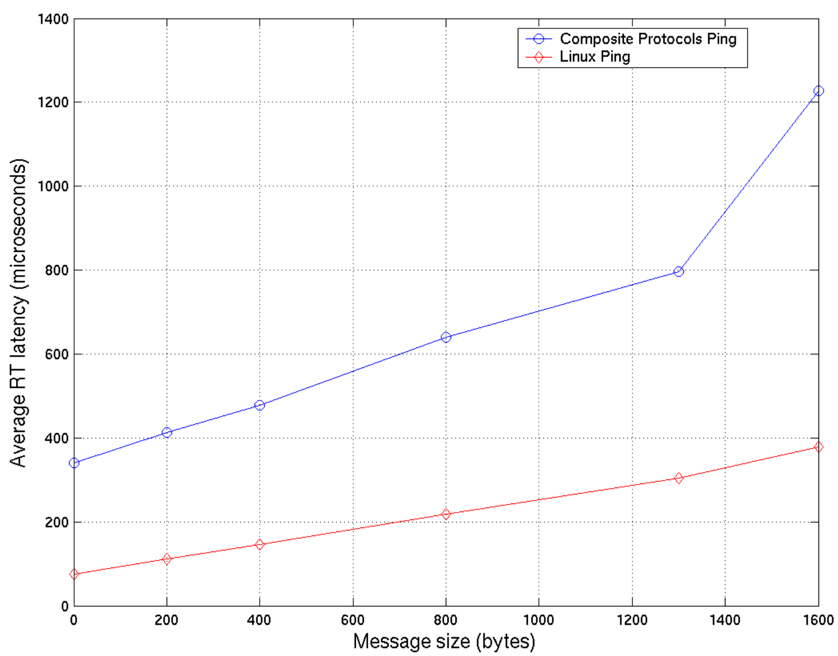
- Linux TCP/IP stack used as benchmark
- Ensemble test applications to mimic *ping* and *ttcp*
- Metrics: Round-trip Latency and One-way Throughput

- Pentium III 533 MHz
- 128 MB RAM, 20 GB HDD
- 1 built-in 100 Mbps NIC
- 1-4 addl. 100 Mbps NICs
- RedHat 7.1, Linux 2.4.3-12
- OCaml v3.06, native code

Test Composite Protocol: UDP-like

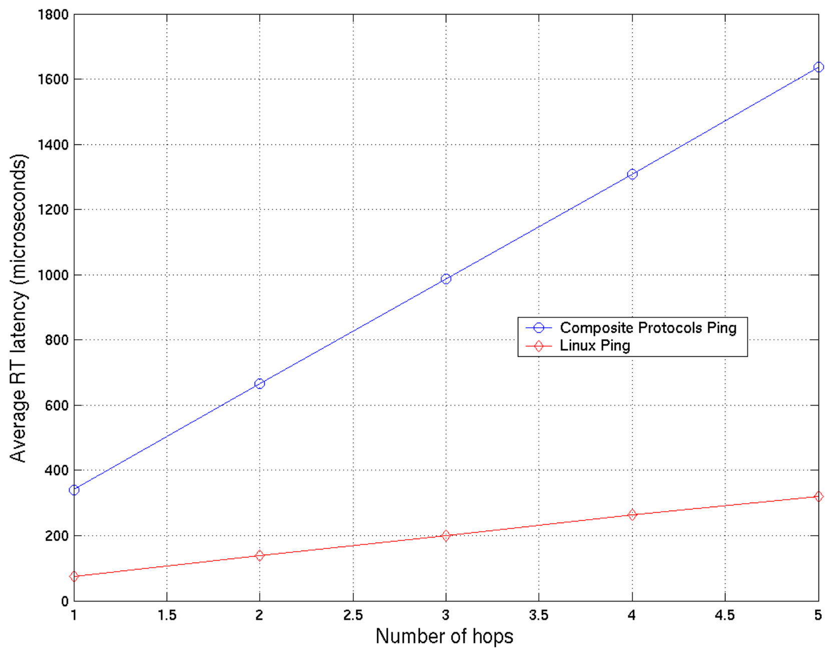


RT Latency vs. Message size



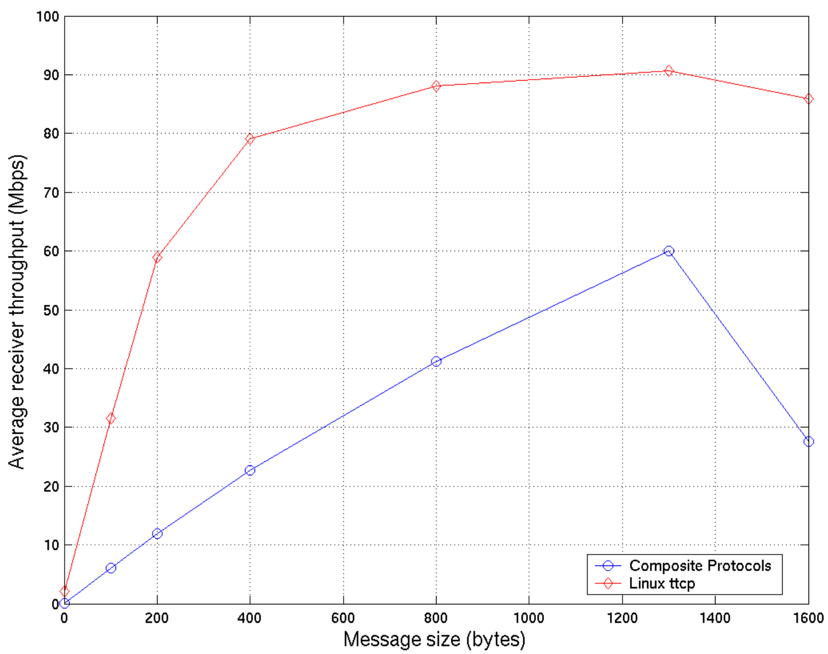
- 11 trials of 1000 messages each
- Machines directly connected
- Standard deviation < 34% of mean
- Sharp increase after 1400 bytes due to fragmentation
- CP ping worse than Linux ping by a factor of 2 to 4
  - SM execution adds overhead
  - Strict layering in framework prevents pointer arithmetic on buffers
  - Ensemble is a user-level program

RT Latency vs. Number of hops



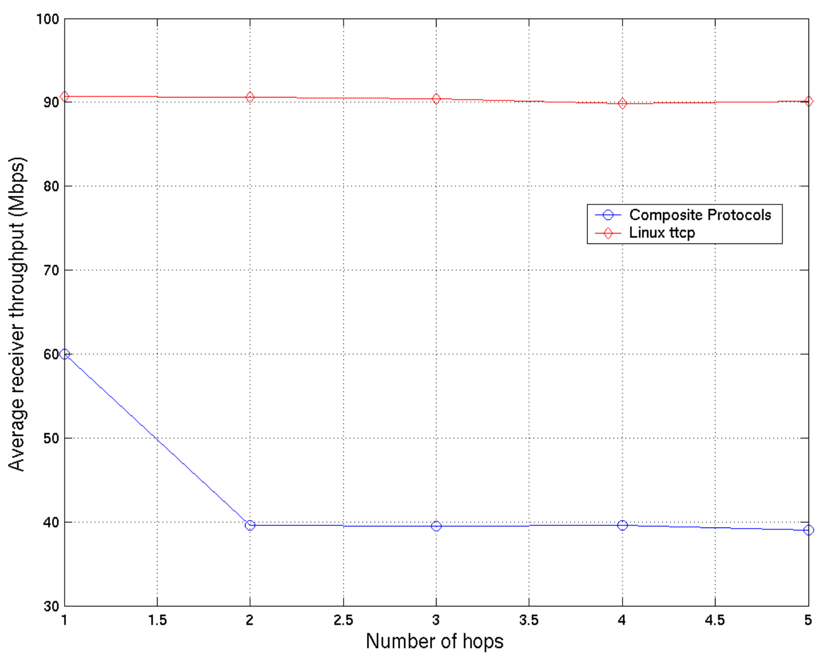
- 11 trials of 1000 messages each
- Minimum message size (1 byte)
- Standard deviation < 40% of mean
- One-hop latency ~ 340 usec
- Each additional hop adds ~ 320 usec
- Per-hop latency increment almost same as one-hop latency due to absence of components above FORWARD at endpoints

Throughput vs. Message size



- 11 trials of 10K messages each
- Machines directly connected
- Standard deviation < 9% of mean
- Sender slowdown factor of 10
- Packet memory overheads
  - 90 bytes by 4 components
  - 28 bytes by framework
- Max. theoretical throughput of 88.04 Mbps at message size of 1354 bytes
- 68% of theoretical max. throughput achieved

Throughput vs. Number of hops



- 11 trials of 10K messages each
- Message size of 1300 bytes
- Standard deviation < 1% of mean
- Sender slowdown factor of 10
- 33% throughput reduction with inclusion of one router, no further reduction with number of routers
- 32% packet loss at first router
- Throughput sustained by a router ~ 39 Mbps

Summary

- New methodology to design modular protocol components
- Highlights
  - State machine representation
  - Emphasis on formal reasoning
  - Explicit memory classification
  - Control interface
- Composite protocol framework implemented over Ensemble
- Components for functional equivalents of IP, UDP, TCP and FTP specified and implemented
- Performance of UDP-like and TCP-like composite protocols evaluated against Linux equivalents

Future Work

- Formal reasoning of protocol component properties
- Tools for enforcing semantic restrictions
  - Completeness of state machines
  - Incompatibilities in SLPM access
- Automatic generation of code from specification
- "Properties-in Protocol-out" configuration tool
- Development of more components
- Efficiency improvements

Questions?