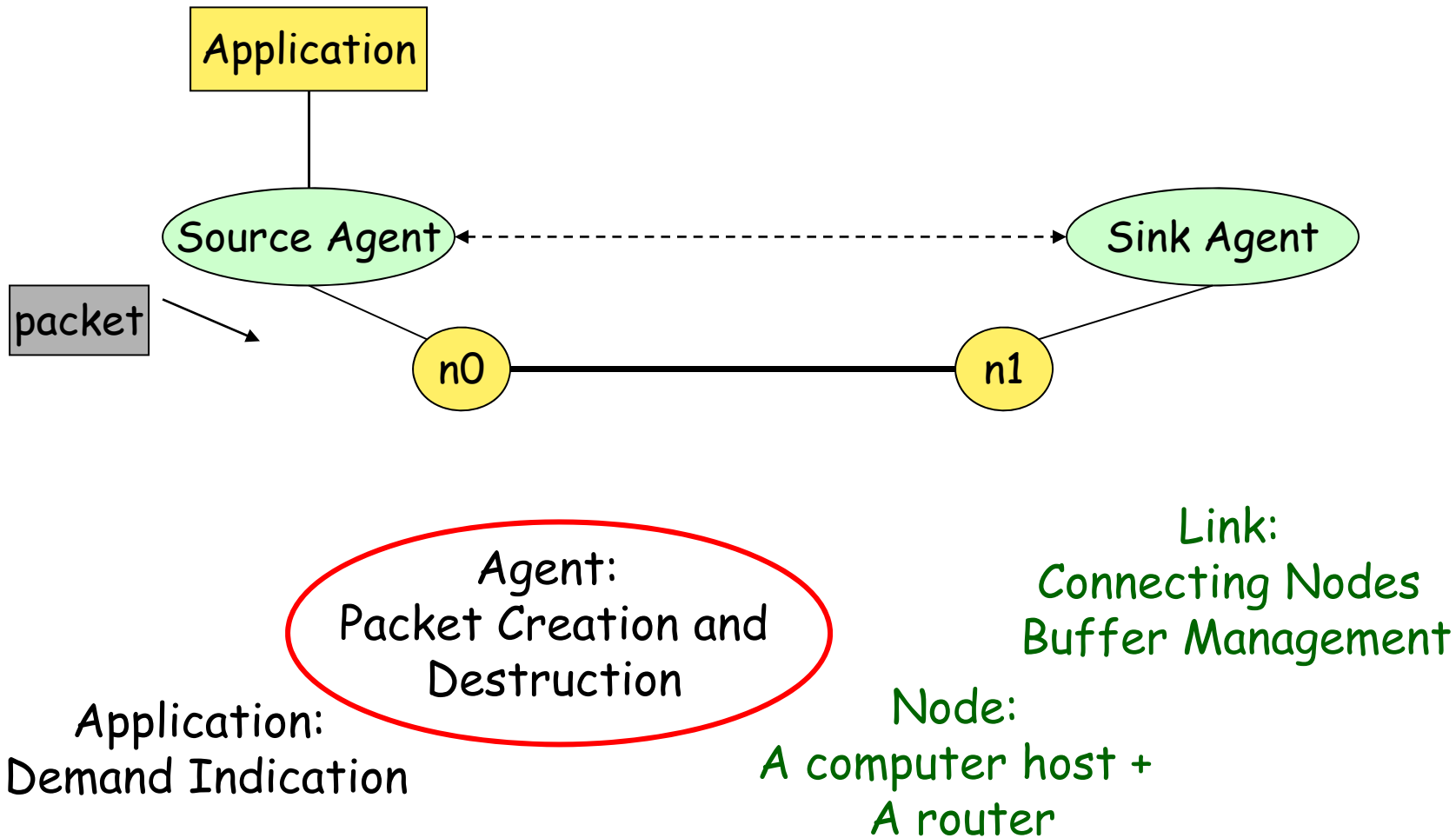


# Transport Control Protocols and UDP Agents



# A Two Node Network



# Outline

- Introduction
- NS2 Transport Layer Agents
- Network Configuration
- UDP Agents
- Summary

# Agent

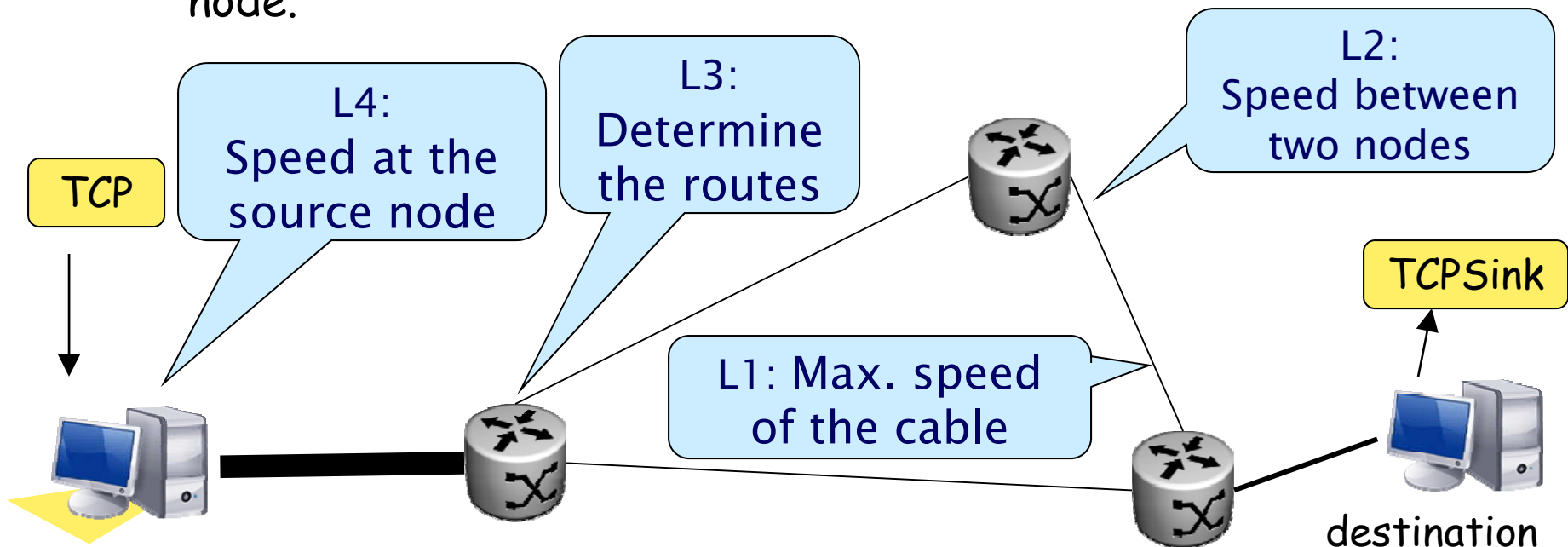
- Creating/destroying packets
- Two main types
  1. Routing agents: Routing packets
  2. Transport layer agents: TCP/UDP packets

Q: what is the main functionality of a transport layer protocol?

( )

# Review of the OSI Model

- L1: Physical layer: Define **max.** speed of a medium.
- L2: Data link layer: Control speed/error between **two connecting nodes**.
- L3: Routing: Find the way to **reach** the destination node.
- L4: Transport Layer: Control speed/error **AT** the **SOURCE** node.



# Transport Layer Protocols

- Encapsulate packets from layers 1, 2, and 3
- End-to-end Flow control
  - At the source node
  - Control the sending rate
  - Regardless of Route, Link-layer protocols, and Physical medium
- End-to-end error control
  - ACK/NACK
  - Packet retransmission



# Outline

- Introduction
- NS2 Transport Layer Agents
- Network Configuration
- UDP Agents
- Summary

# Transport Layer Agents

- Application: Create **user demand** to send data
- Simulator: Indicate the **source and destination**
- Sending Agent
  - Input
    - **Src/Dst** indicated by the Simulator
    - **User demand** indicated by the Application
  - **Create** packets
  - **Send out** packets to lower layer network
  - Control the **rate** at which the packets are sent.
- Receiving Agents
  - **Destroy** packets
  - Send out **acknowledgement**



# C++ and OTcl Implementation

- C++ and OTcl class Agent
- C++ class Agent derives from class Connector
- Base class for all Agent
  - UdpAgent  $\leftrightarrow$  Agent / UDP ,
  - TcpAgent  $\leftrightarrow$  Agent / TCP



# C++ and OTcl Variables

C++ Type	C++ variable	OTcl instvar	Description
ns_addr_t	here_		
	here_.addr_	agent_addr_	Address of the attached node
	here_.port_	agent_port_	Port where the agent is attached
ns_addr_t	dst_		
	dst_.addr_	dst_addr_	Address of the node attaching to a peering agent
	dst_.port_	dst_port_	Port where the peering agent is attached
int	size_	N/A	Packet size
packet_t	type_	N/A	Packet type
int	seqno_	N/A	Current sequence number

# C++ and OTcl Variables

C++ Type	C++ variable	OTcl instvar	Description
int	fid_	fid_	Flow ID
int	prio_	prio_	IPv6 priority field (e.g., 0 = unspecified, 1 = background traffic)
int	flags_	flags_	Flags
int	defttl_	ttl_	Default time to live value
Application*	app_	N/A	A pointer to an application
int	uidcnt_	N/A	Total number of packets generated by all agents

# C++ Implementation

Function	Meaning
<code>recv(p,h)</code>	Receives a packet <code>p</code> .
<code>send(p,h)</code>	Sends a packet <code>p</code> to the attached Application.
<code>send(nbytes)</code>	Sends a message with <code>nbytes</code> bytes to the attached Application.
<code>sendmsg(nbytes)</code>	Sends a message with <code>nbytes</code> bytes to the attached Application.
<code>timeout(tno)</code>	Actions to be performed at timeout
<code>connect(dst)</code>	Connects to a dynamic destination <code>dst</code> .
<code>close()</code>	Closes a connection-oriented session .
<code>listen()</code>	Waits for a connection-oriented session .

# C++ Implementation

Function	Meaning
<code>attachApp(app)</code>	Stores <code>app</code> in variable <code>app_</code> .
<code>allocpkt()</code>	Creates a packet.
<code>initpkt(p)</code>	Initializes the input packet <code>p</code> .
<code>recvBytes(bytes)</code>	Sends data of <code>bytes</code> bytes to the attached application.
<code>idle()</code>	Tells the application that the agent has nothing to transmit.

# C++ Implementation

- Function `sendmsg(nbytes)`
  - Invoked by application to send a message
  - Implemented in the derived class

- Function `recv(p, h)`

```
void Agent::recv(Packet* p, Handler*)
{
    if (app_)
        app_->recv(hdr_cmn::access(p)->size());
    Packet::free(p);
}
```

- Function `attachApp(app)`

```
void Agent::attachApp(Application *app)
{
    app_ = app;
}
```

# C++ Implementation

- Function `allocpkt()`

```
Packet* Agent::allocpkt() const
{
    Packet* p = Packet::alloc();
    initpkt(p);
    return (p);
}
```

- Function `initpkt(p)`

```
void Agent::initpkt(Packet* p)
{
    hdr_cmn* ch = hdr_cmn::access(p);
    ch->uid() = uidcnt_++;
    ch->ptype() = type_;
    ...
    hdr_ip* iph = hdr_ip::access(p);
    iph->saddr() = here_.addr_;
    iph->sport() = here_.port_;
    iph->daddr() = dst_.addr_;
    iph->dport() = dst_.port_;
    ...
}
```



# A Guide for Creating a New Agent

1. Define the **hierarchy**: Based/derived classes
2. Define C++ and OTcl class **variables**
3. Define the **constructor** in both the hierarchy (**bind** the C++/OTcl variables here)
4. Implement the following **key functions**  
`sendmsg(nbyte), recv(p,h), timeout(tno)`
5. Define **OTcl commands**
6. Define **timer** (if necessary)



# Outline

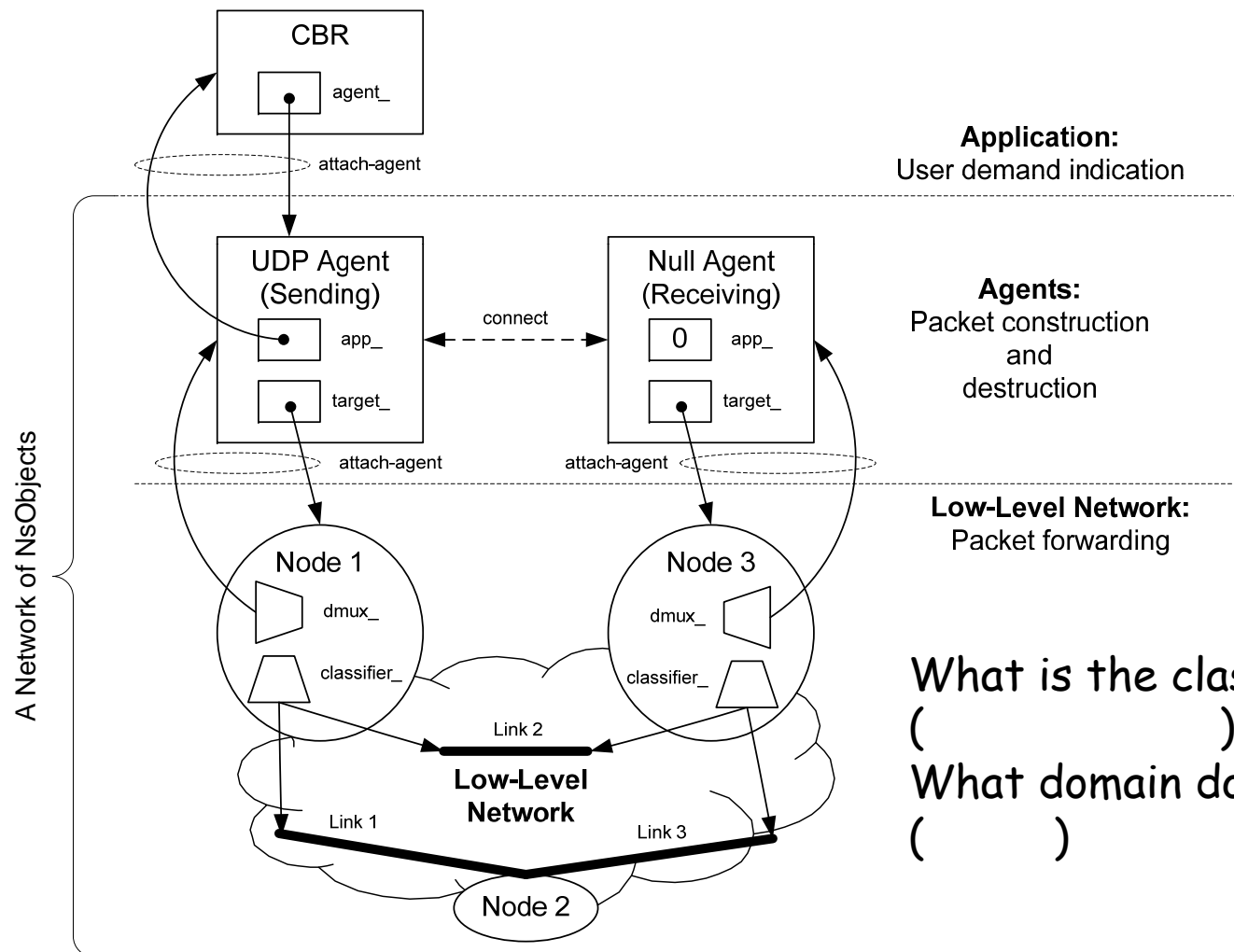
- Introduction
- NS2 Transport Layer Agents
- Network Configuration
- UDP Agents
- Summary

# Network Configuration

- 4 Steps in an OTcl simulation script
  1. **Create** a sending agent, a receiving agent, and an application
  2. Connect the agents to the **applications**
  3. Connect the agents to the **low-level network**
  4. Associating **sending and receiving agents**



# Applications, Agents, and Low-level Networks



# OTcl simulation script

```
set ns [new Simulator]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 5Mb 2ms DropTail
$ns duplex-link $n1 $n3 5Mb 2ms DropTail
```

```
#=== UDP-Null peering starts here ===
set udp [new Agent/UDP]
set null [new Agent/Null]
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$ns attach-agent $n1 $udp
$ns attach-agent $n3 $null
$ns connect $udp $null
```

Focus here!!

Step1: The creation

Step2: Agent ⇔ Application

Step3: Agent ⇔ Network

Step4: Sending → Receiving

# Network Configuration

## Step1: The Creation

- An Agent instance: `new Agent/<type>`
- An Application instance: `new Application/<type>`

## Step2: Agent ↔ Application

- Use OTcl command of class Application

`$app attach-agent $agent`

```
//~ns/apps/app.cc
int Application::command(int argc, const
char*const* argv){
    if (strcmp(argv[1], "attach-agent") == 0) {
        agent_ = (Agent*) TclObject::lookup(argv[2]);
        agent_>attachApp(this);
        return(TCL_OK);
    }
}
```

```
//~ns/common/agent.cc
void Agent::attachApp(Application *app)
{
    app_ = app;
}
```

# Network Configuration

## Step3: Agent $\leftrightarrow$ Network

- Use OTcl command of class Simulator (see Chapter 6: Node)

```
$ns attach-agent $node $agent
```

1. `$ns attach-agent $node $agent`
2. `$node attach $agent {port ""}`
3. `$node add-target $agent $port`
4. `$rm attach $agent $port`
5. `$agent target [[ $self node ] entry]`
5. `[[ $self node ] demux] install $port $agent`

# Network Configuration

## Step4: Sending ↔ Receiving

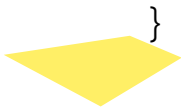
- Use an OTcl command of class Simulator

```
$ns connect $s_agent $r_agent
```

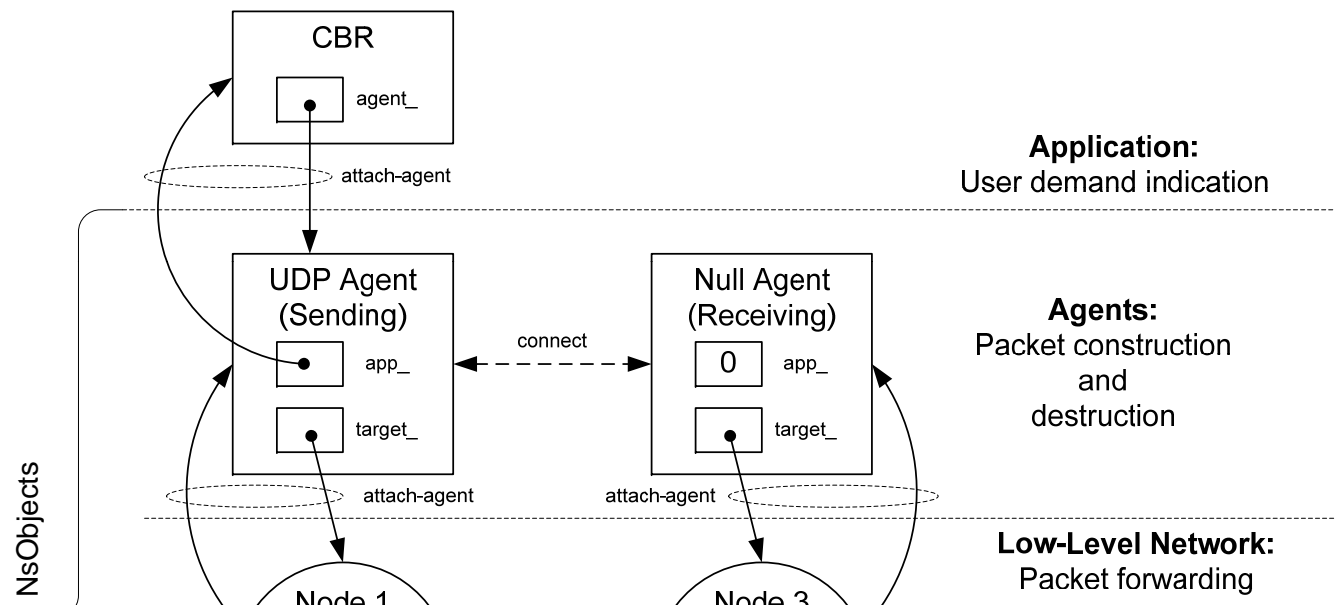
```
//~/ns/tcl/lib/ns-lib.tcl
Simulator instproc connect {src dst} {
    ...
    $self simplex-connect $src $dst
    $self simplex-connect $dst $src
}

Simulator instproc simplex-connect { src dst } {
    ...
    $src set dst_addr_ [$dst set agent_addr_]
    $src set dst_port_ [$dst set agent_port_]
}
```

When a packet is created ports and addresses of the source and destination are stored in the packet



# Applications, Agents, and Low-level Networks



	Sending agent	Receiving agent
Upstream - object - packet forwarding function	Application sendmsg	Node recv
Downstream object - object - packet forwarding function	Node recv	N/A N/A

# Outline

- Introduction
- NS2 Transport Layer Agents
- Network Configuration
- UDP Agents
- Summary

# User Datagram Protocol

- Transport layer protocol
- Unreliable and connectionless
- RFC 1122  
(<http://www.networksorcery.com/enp/protocol/udp.htm>)
- Main functionality
  - Mux/Demux data flows in source/destination nodes
  - Put the address/port of source/destination node in packets, and
  - Forward to the packet to IP layer.

# UDP: NS2 Implementation

- Two types of Agents

## 1. Sending Agent

- OTcl: Agent /UDP  $\Leftrightarrow$  C++: UdpAgent

## 2. Receiving Agent

- OTcl: Agent /Null  $\Leftrightarrow$  C++: **None**

- See `~ns/tcl/lib/ns-agent.tcl`



# A Guide for Creating a New Agent

1. Define the hierarchy: Based/derived classes
2. Define C++ and OTcl class variables
3. Define the constructor in both the hierarchy (bind the C++/OTcl variables here)
4. Implement the following key functions  
`sendmsg(nbyte), recv(p,h), timeout(tno)`
5. Define OTcl commands
6. Define timer (if necessary)



# UDP Sending Agent

## Step1: Define class inheritance

- C++: Agent → UdpAgent
- OTcl: Agent → Agent/UDP
- Binding C++ and OTcl classes

```
//~/ns/apps/udp.cc
static class UdpAgentClass : public TclClass {
public:
    UdpAgentClass() : TclClass("Agent/UDP") {}
    TclObject* create(int, const char*const*) {
        return (new UdpAgent());
    }
} class_udp_agent;
```

What is the  
difference between  
UdpAgent::seqno\_  
and Packet::uid\_?

## Step2: Define variables

- C++: UdpAgent::seqno\_

# UDP Sending Agent

## Step3: Define the constructors

### - C++:

```
//~/ns/apps/udp.cc
UdpAgent::UdpAgent() : Agent(PT_UDP), seqno_(-1){
    bind("packetSize_", &size_);
}
//~/ns/common/agent.cc
Agent::Agent(packet_t pkttype):
    size_(0), type_(pkttype), app_(0){}
```

### - OTcl: None

# A Guide for Creating a New Agent

1. Define the hierarchy: Based/derived classes
2. Define C++ and OTcl class variables
3. Define the constructor in both the hierarchy (bind the C++/OTcl variables here)
4. Implement the following key functions  
`sendmsg(nbyte), recv(p,h), timeout(tno)`
5. Define OTcl commands
6. Define timer (if necessary)



# UDP Sending Agent

## Step4: Define C++ functions

- `recv(p, h):`
  - Main packet reception function
  - Invoked by: an underlying NSObject
  - What it does:
    1. Tell the application of the amount of data received.
    2. Destroy the packet

```
void Agent::recv(Packet* p, Handler*)
{
    if (app_)
        app_>recv(hdr_cmn::access(p)->size());
    Packet::free(p);
}
```

# UDP Sending Agent

- `sendmsg(nbytes, data, flags):`
  - Send out `nbytes` bytes of data
  - Invoked by: the attached application
  - What it does: Create and send out the packets

```
//~/ns/apps/udp.cc
virtual void sendmsg(int nbytes, const char *flags = 0){
    sendmsg(nbytes, NULL, flags);
}
void UdpAgent::sendmsg(int nbytes, AppData* data, const char* flags)
{
    Packet *p; int n = nbytes / size_;
    while (n-- > 0) {
        p = allocpkt(); hdr_cmn::access(p)->size() = size_;...
        target_->recv(p);
    }
    ...
    idle();
}
```

Textbook: T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*, Springer 2008.

# UDP Sending Agent

## Step5: Define OTcl commands and Instprocs

- Two key Otcl commands: `send{nbytes, str}`, `sendmsg{nbytes, str, flags}`
- Send `nbytes` bytes of packets whose content is `str`
- Make use of  
`UdpAgent::sendmsg(nbytes, str, flag)`
- Not usually used
- Usually: Application C++ object invokes  
`UdpAgent::sendmsg(nbytes)`



# UDP: NS2 Implementation

- Two types of Agents

## 1. Sending Agent

- OTcl: Agent /UDP  $\Leftrightarrow$  C++: UdpAgent

## 2. Receiving Agent

- OTcl: Agent /Null  $\Leftrightarrow$  C++: None

- See `~ns/tcl/lib/ns-agent.tcl`

- Associating Sending and Receiving Agents

# UDP Receiving Agent

Step1: Define class inheritance

- C++: None
- OTcl: Agent → Agent/Null

Step2: Define variables: None

Step3: Define the constructor: None (use that of Agent)

Step4: Define key function: None (use that of Agent)

```
void Agent::recv(Packet* p, Handler*)
{
    if (app_)
        app_>recv(hdr_cmn::access(p)->size());
    Packet::free(p);
}
```

Step5: Define OTcl commands and Instprocs: None

Step6: Define Timer: None

# Outline

- Introduction
- NS2 Transport Layer Agents
- Network Configuration
- UDP Agents
- Summary

# Summary

- Main functionality

- Application ( )
- Simulator ( )
- Agent
  - 1)
  - 2)
  - 3)

- UDP agents

- Sender ( )
- Receiver ( )

