Graphite: Extending Models

How to add a new network model in Graphite

Overview

- How to implement a new model in Graphite
- Example: add a network model
  - Based on NetworkModelEmeshHopCounter, included in github distribution
- Other models follow similar patterns
- Code listings aren't exact, but closely match code in repository

Adding a Model

- Implement base class

- Register new model

Adding a Network Model

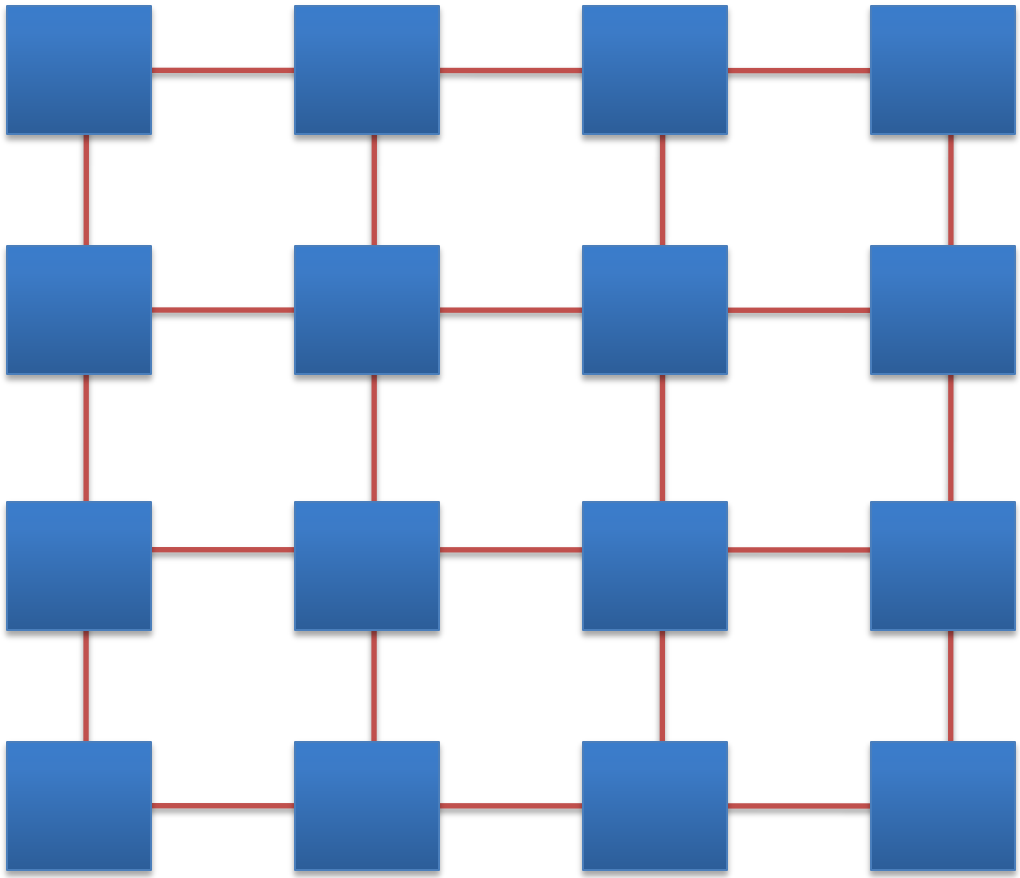
- Implement base class
  - **NetworkModel**

- Register new model
  - Add type to **network_types.h**
  - Add to factory function in **network_model.cc**
  - Use model in **carbon_sim.cfg**

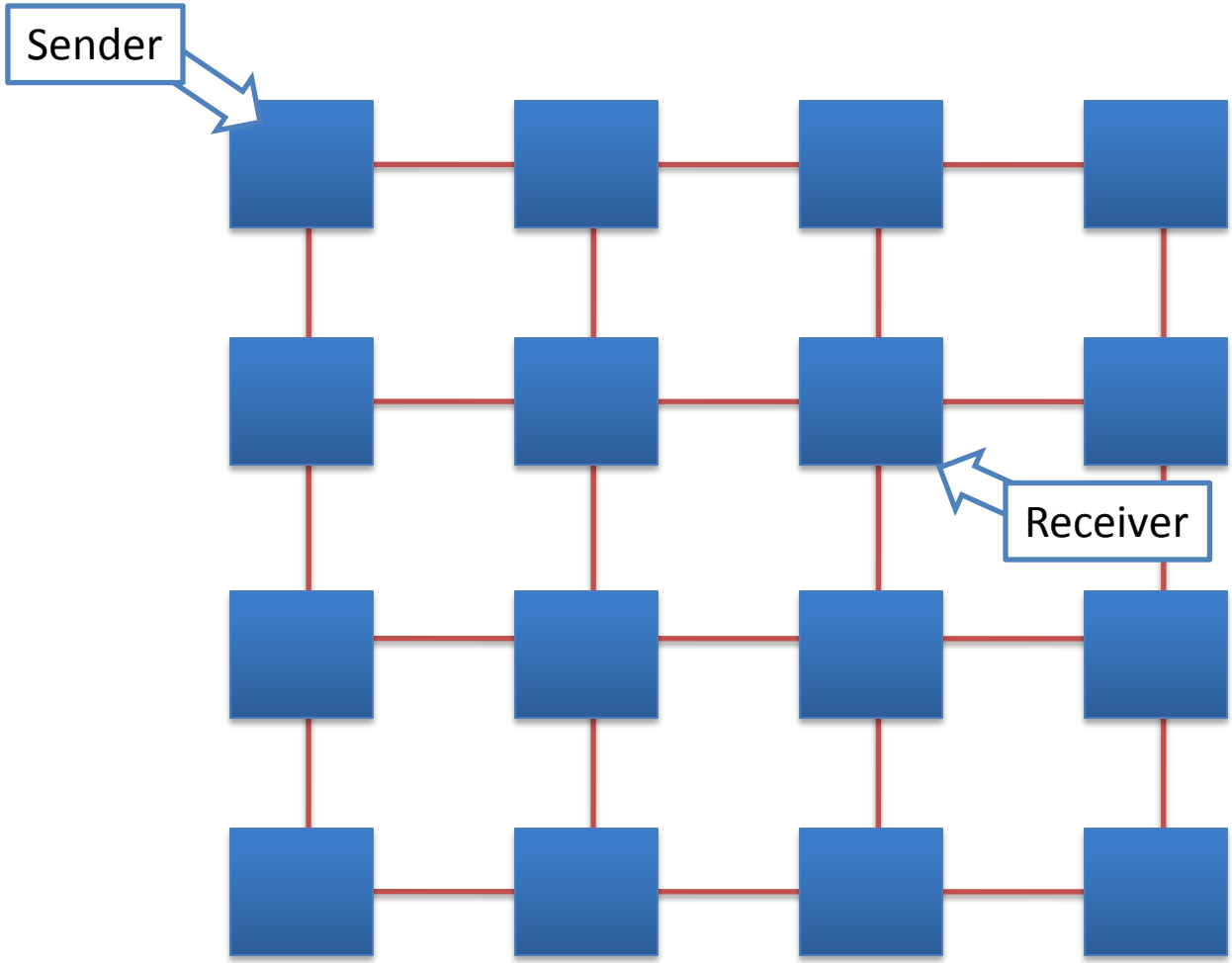
HopCounter

- New network model
- Counts number of network hops to destination on a mesh
  - Assumes one cycle per hop
  - Add additionally serialization latency
- Delivers packet to destination immediately
- Properties
  - Fast: no extra communication and immediate delivery
  - Inaccurate: no contention modeling

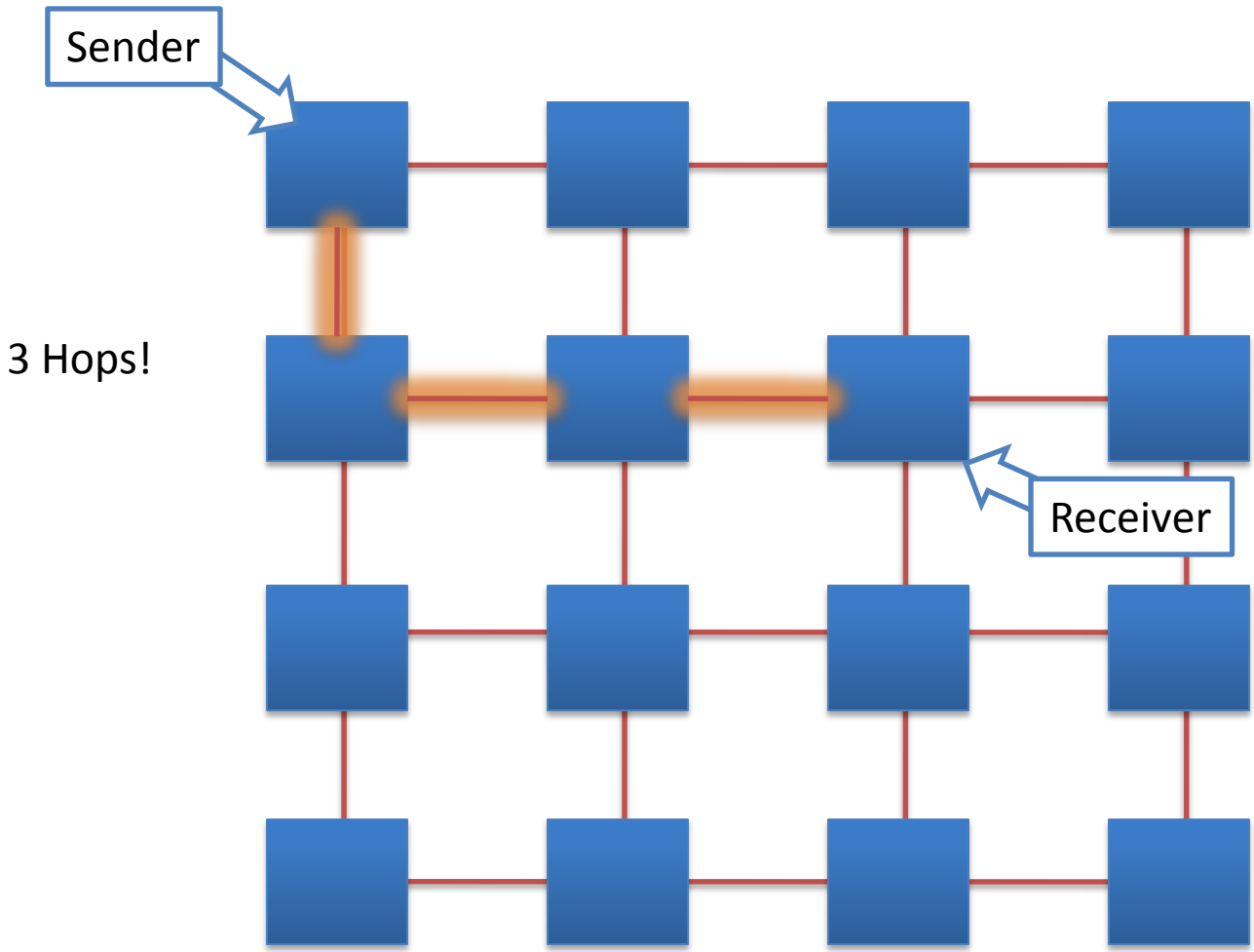
HopCounter Latency



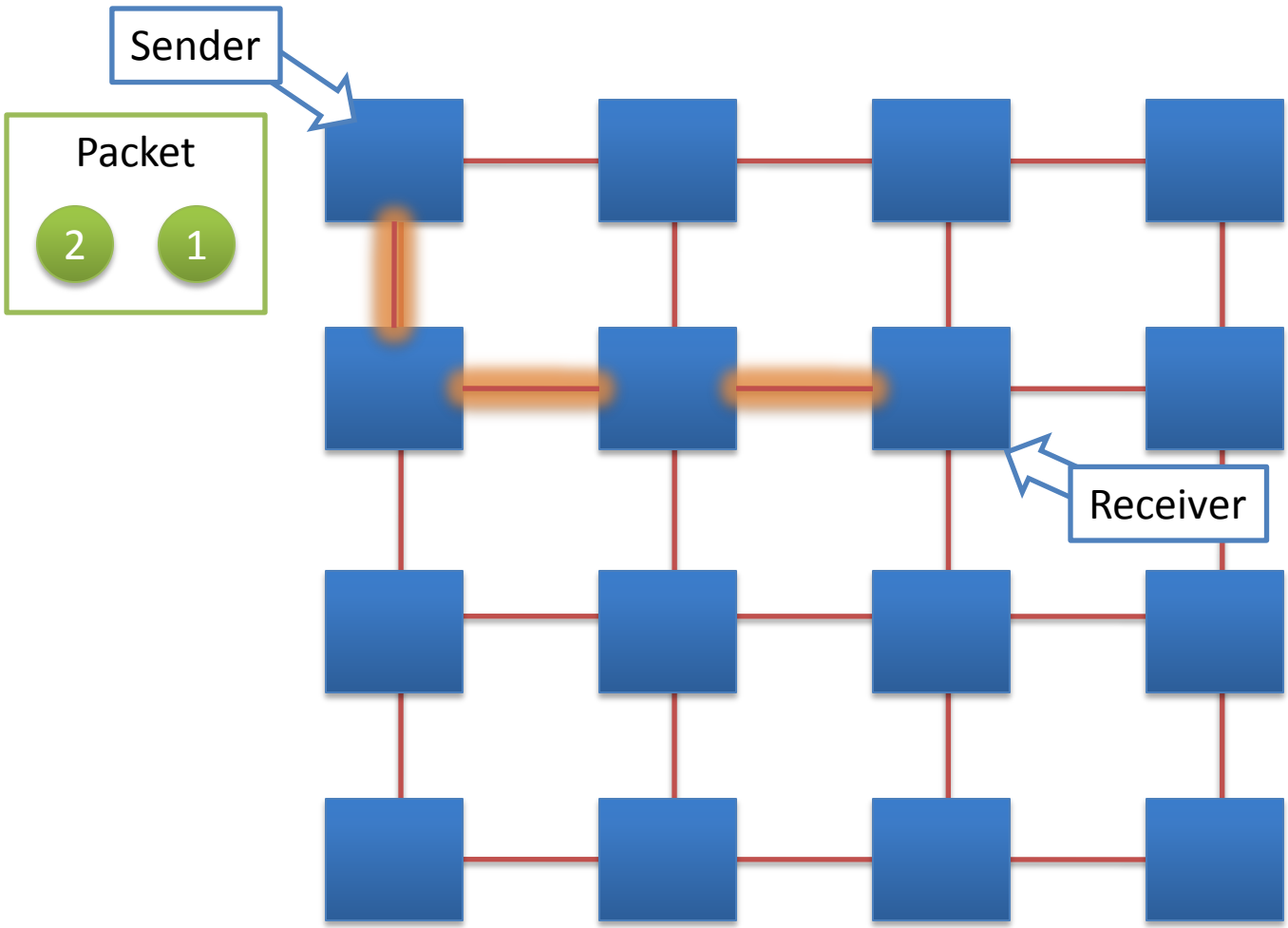
HopCounter Latency



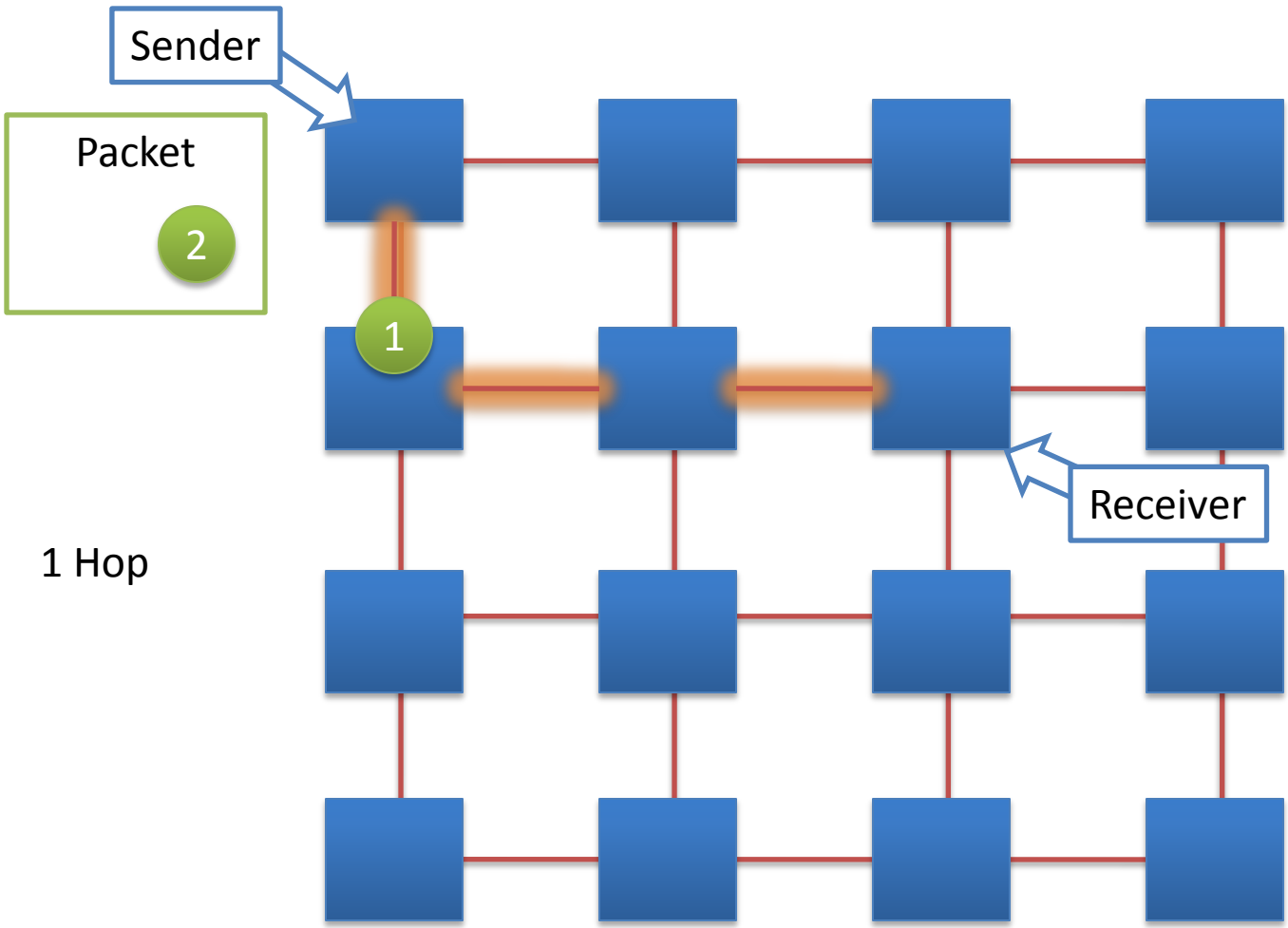
HopCounter Latency



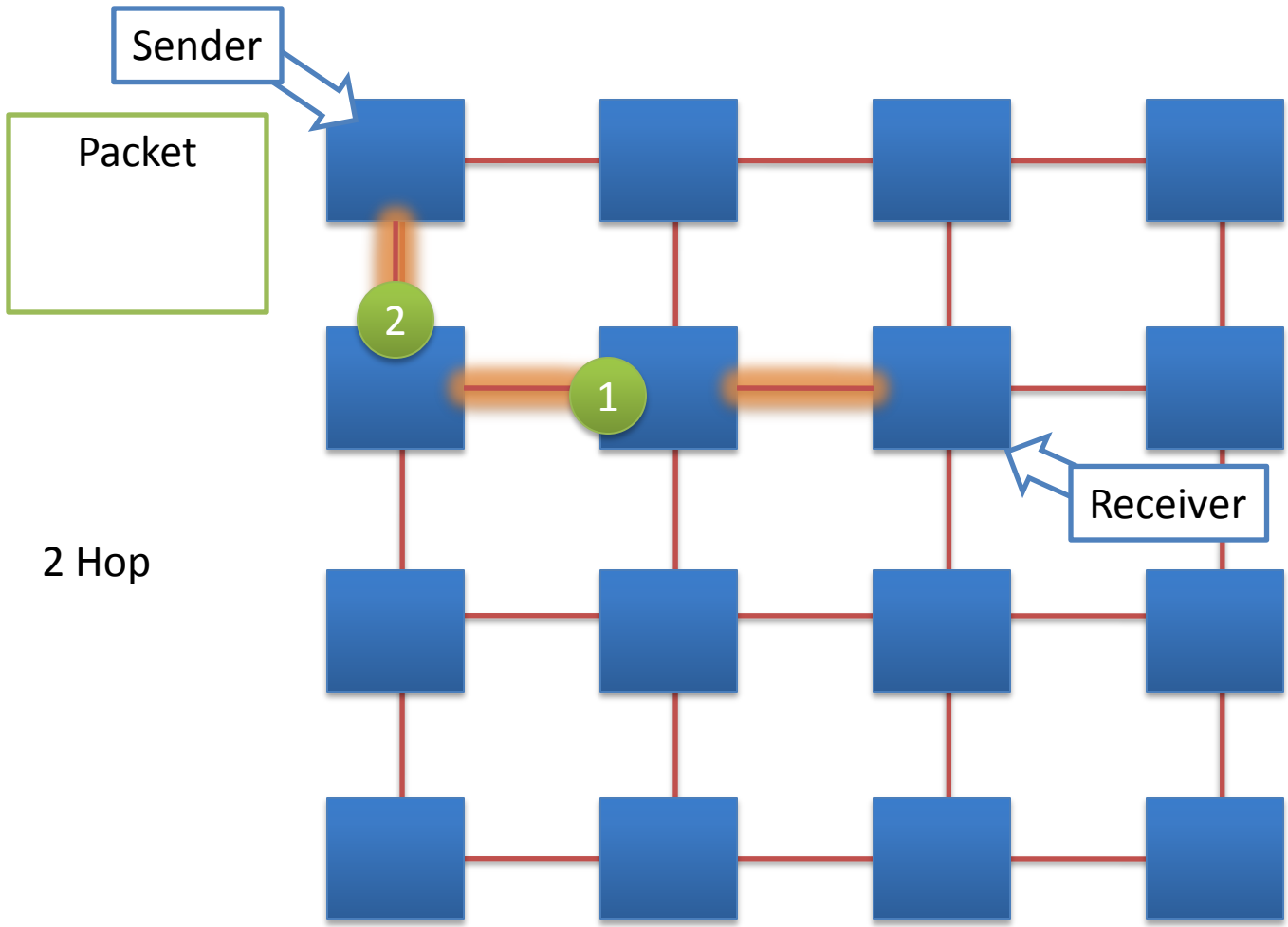
HopCounter Latency



HopCounter Latency

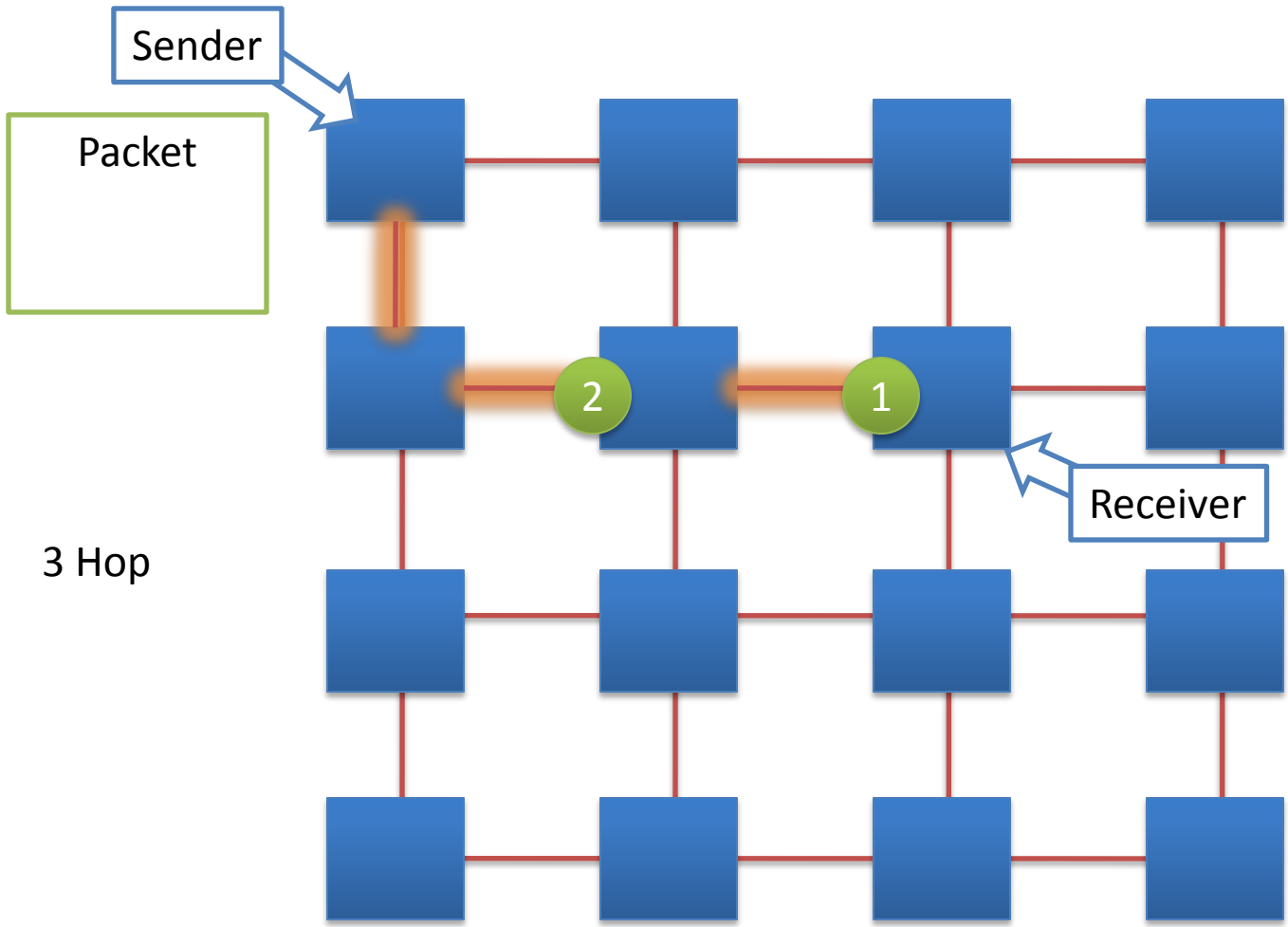


HopCounter Latency

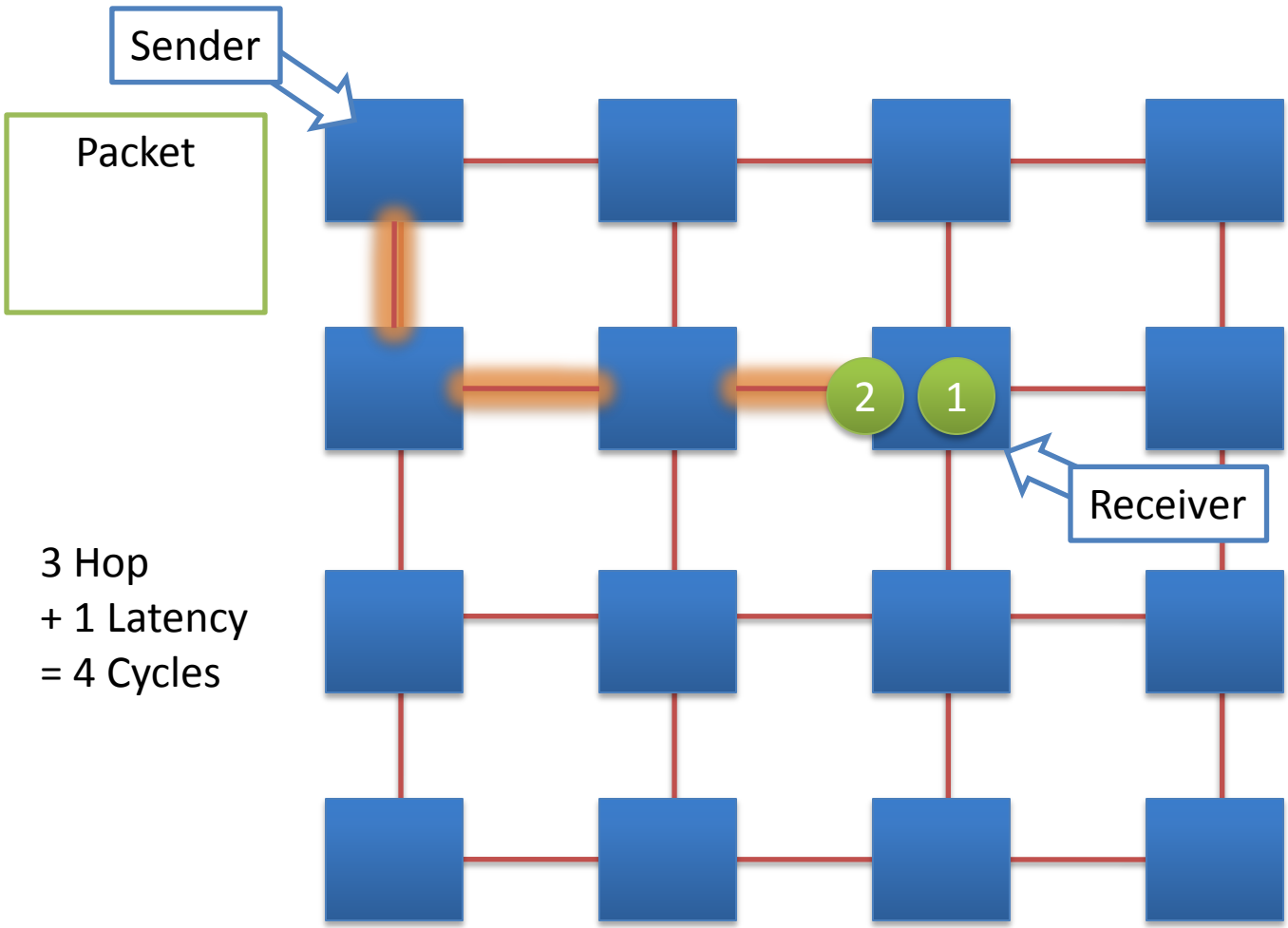


2 Hop

HopCounter Latency

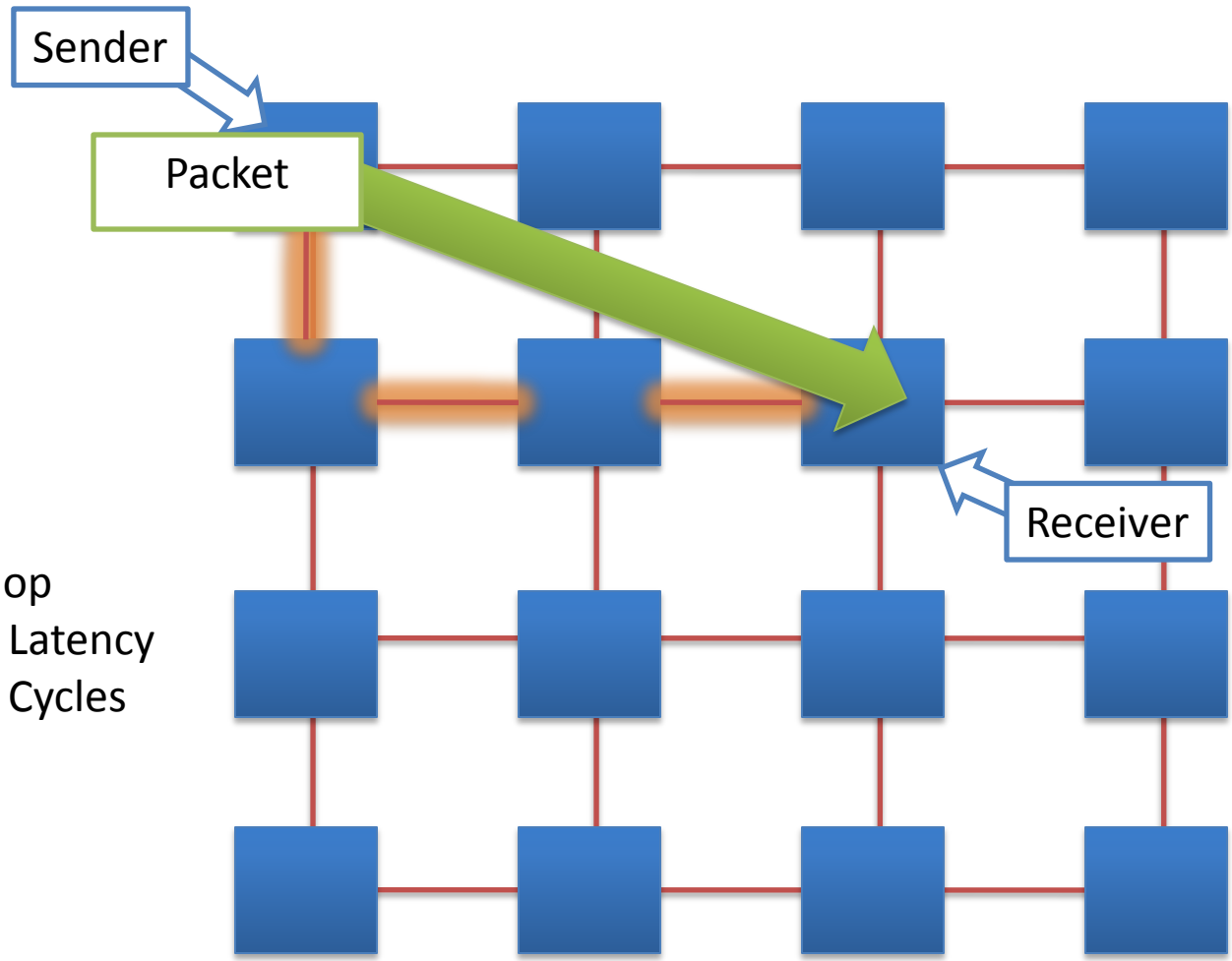


HopCounter Latency



3 Hop
+ 1 Latency
= 4 Cycles

HopCounter Latency



3 Hop
+ 1 Latency
= 4 Cycles

NetworkModel Interface

- void **routePacket**(NetPacket pkt,
  vector<Hop> &hops)
  - Takes a new packet as input
  - Routes the packet to its next hop(s) on the network
  - Updates timestamp

- void **processReceivedPacket**(NetPacket pkt)
  - Hook to apply additional processing to an incoming packet

Important Structures

- /common/network/network.h

- **NetPacket**
  - A single piece of data moving in the network
  - **time** – The timestamp for this packet (uint64_t)
  - **type** – The packet type; determines which network to use
  - **sender** – Index of sending core
  - **receiver** – Index of receiving core
  - **data**, **length** – Bytes to send

- **Hop**
  - Represents a single hop on the network for routing
  - **final_dest** – The final destination of the packet (core ID)
  - **next_dest** – The next destination of the packet (core ID)
  - **time** – Arrival time for this packet at its next hop (uint64_t)

HopCounter Declaration

```
class HopCounter : public NetworkModel
{
public:
  HopCounter(Network *net);
  ~HopCounter();

  void routePacket(const NetPacket &pkt,
                   std::vector<Hop> &nextHops);
  void processReceivedPacket(NetPacket &pkt);

private:

  void computePosition(core_id_t core, SInt32 &x, SInt32 &y);
  SInt32 computeDistance(SInt32 x1, SInt32 y1, SInt32 x2, SInt32 y2);

  UInt64 computeSerializationLatency(UInt32 pkt_length);
};
```

HopCounter Declaration

```
class HopCounter : public NetworkModel
{
public:
  HopCounter(Network *net);
  ~HopCounter();

  void routePacket(const NetPacket &pkt,
                   std::vector<Hop> &nextHops);
  void processReceivedPacket(NetPacket &pkt);

private:

  void computePosition(core_id_t core, SInt32 &x, SInt32 &y);
  SInt32 computeDistance(SInt32 x1, SInt32 y1, SInt32 x2, SInt32 y2);

  UInt64 computeSerializationLatency(UInt32 pkt_length);
};
```

HopCounter Declaration

```
class HopCounter : public NetworkModel
{
public:
  HopCounter(Network *net);
  ~HopCounter();

  void routePacket(const NetPacket &pkt,
                   std::vector<Hop> &nextHops);
  void processReceivedPacket(NetPacket &pkt);

private:

  void computePosition(core_id_t core, SInt32 &x, SInt32 &y);
  SInt32 computeDistance(SInt32 x1, SInt32 y1, SInt32 x2, SInt32 y2);

  UInt64 computeSerializationLatency(UInt32 pkt_length);
};
```

routePacket

```
void HopCounter::routePacket(const NetPacket &pkt,
                             std::vector<Hop> &nextHops)
{
  UInt64 serialization_latency =
    computeSerializationLatency(pkt.length);

  SInt32 sx, sy, dx, dy;

  computePosition(pkt.sender, sx, sy);
  computePosition(pkt.receiver, dx, dy);
  UInt64 latency = computeDistance(sx, sy, dx, dy);

  if (pkt.receiver != pkt.sender)
    latency += serialization_latency;

  Hop h;
  h.final_dest = pkt.receiver;
  h.next_dest = pkt.receiver;
  h.time = pkt.time + latency;

  nextHops.push_back(h);
}
```

routePacket

```
void HopCounter::routePacket(const NetPacket &pkt,
                             std::vector<Hop> &nextHops)
{
  UInt64 serialization_latency =
    computeSerializationLatency(pkt.length);

  SInt32 sx, sy, dx, dy;

  computePosition(pkt.sender, sx, sy);
  computePosition(pkt.receiver, dx, dy);
  UInt64 latency = computeDistance(sx, sy, dx, dy);

  if (pkt.receiver != pkt.sender)
    latency += serialization_latency;

  Hop h;
  h.final_dest = pkt.receiver;
  h.next_dest = pkt.receiver;
  h.time = pkt.time + latency;

  nextHops.push_back(h);
}
```

- Compute the length of the packet and serialization latency
- Based on network width
  - Model parameter, not property of Network class

routePacket

```
void HopCounter::routePacket(const NetPacket &pkt,
                             std::vector<Hop> &nextHops)
{
  UInt64 serialization_latency =
    computeSerializationLatency(pkt.length);

  SInt32 sx, sy, dx, dy;

  computePosition(pkt.sender, sx, sy);
  computePosition(pkt.receiver, dx, dy);
  UInt64 latency = computeDistance(sx, sy, dx, dy);

  if (pkt.receiver != pkt.sender)
    latency += serialization_latency;

  Hop h;
  h.final_dest = pkt.receiver;
  h.next_dest = pkt.receiver;
  h.time = pkt.time + latency;

  nextHops.push_back(h);
}
```

- Compute latency of packet
- Based on # of hops in network
- Add serialization latency (virtual cut-through)

routePacket

```
void HopCounter::routePacket(const NetPacket &pkt,
                             std::vector<Hop> &nextHops)
{
  UInt64 serialization_latency =
    computeSerializationLatency(pkt.length);

  SInt32 sx, sy, dx, dy;

  computePosition(pkt.sender, sx, sy);
  computePosition(pkt.receiver, dx, dy);
  UInt64 latency = computeDistance(sx, sy, dx, dy);

  if (pkt.receiver != pkt.sender)
    latency += serialization_latency;

  Hop h;
  h.final_dest = pkt.receiver;
  h.next_dest = pkt.receiver;
  h.time = pkt.time + latency;

  nextHops.push_back(h);
}
```

- Route packet to final destination
  - A “hop-by-hop” model would send it to an intermediate node in mesh (h.final_dest != h.next_dest)
- Add latency to final timestamp
- Add Hop to nextHop vector
  - Broadcast/multicast handling omitted; packets would be sent as multiple unicast messages
  - I.e., nextHops would contain multiple destinations

processReceivedPacket

```
void
HopCounter::processReceivedPacket(NetPacke
      t &pkt)
{
  ++_numPacketsReceived;
}
```

- This function is used mostly to track statistics
- For example, HopCounter uses it to track the number of packets received

Register New Model

/common/network/network_types.h

```
#ifndef NETWORK_TYPES_H
#define NETWORK_TYPES_H

enum NetworkType
{
  NETWORK_MAGIC,
  NETWORK_HOP_COUNTER,
  NETWORK_ANALYTICAL_MESH,
  NETWORK_EMESH_HOP_BY_HOP_BASIC,
  NUM_NETWORK_TYPES
};

#endif // NETWORK_TYPES_H
```

- Add type to list of valid network types

Register New Model (cont'd)

/common/network/network_model.cc

```
NetworkModel*
NetworkModel::createModel(Network *net, UInt32 model_type,
        EStaticNetwork net_type)
{
  switch (model_type)
  {
  case NETWORK_MAGIC:
    return new NetworkModelMagic(net);

  case NETWORK_HOP_COUNTER:
    return new HopCounter(net);

  case NETWORK_ANALYTICAL_MESH:
    return new NetworkModelAnalytical(net, net_type);

  case NETWORK_EMESH_HOP_BY_HOP_BASIC:
    return new NetworkModelEMeshHopByHopBasic(net);

  default:
    assert(false);
    return NULL;
  }
}
```

- Add to factory function in NetworkModel

Register New Model (cont'd)

/common/network/network_model.cc

```
UInt32
NetworkModel::parseNetworkType(string str)
{
  if (str == "magic")
    return NETWORK_MAGIC;
  else if (str == ”hop_counter")
    return NETWORK_HOP_COUNTER;
  else if (str == "analytical")
    return NETWORK_ANALYTICAL_MESH;
  else if (str == "emesh_hop_by_hop_basic")
    return NETWORK_EMESH_HOP_BY_HOP_BASIC;
  else if (str ==
      "emesh_hop_by_hop_broadcast_tree")
    return
      NETWORK_EMESH_HOP_BY_HOP_BROADCAST_
      TREE;
  else
    return (UInt32)-1;
}
```

- Give it a human-readable name

Use It!

/carbon_sim.cfg

…

This describes the various models used for the different networks on the core

[network]

user_model_1 = **hop_counter**

user_model_2 = **hop_counter**

memory_model_1 = **hop_counter**

memory_model_2 = **hop_counter**

system_model = magic

…

- Use human-readable name in configuration file
- Set one of the static networks to use this model
  - Different networks for different classes of traffic
  - I.e., memory traffic, user messages, system communication, etc..

Considerations

- Correctness
  - Some models affect simulation behavior (e.g., routing packets)
  - Must be done correctly!
- Performance/accuracy tradeoffs
  - Do you need hop-by-hop modeling, or can you skip intermediate nodes?
    - 2.5x performance difference with 64 target cores
- How to manage sharing
  - Centralize at MCP (e.g., bus model)
  - Distributed (e.g., hop-by-hop mesh model)
  - Pure local (e.g., hop counter mesh model)

Summary

- Other models follow similar patterns, but with different interfaces
- Adding a new model isn’t too painful!