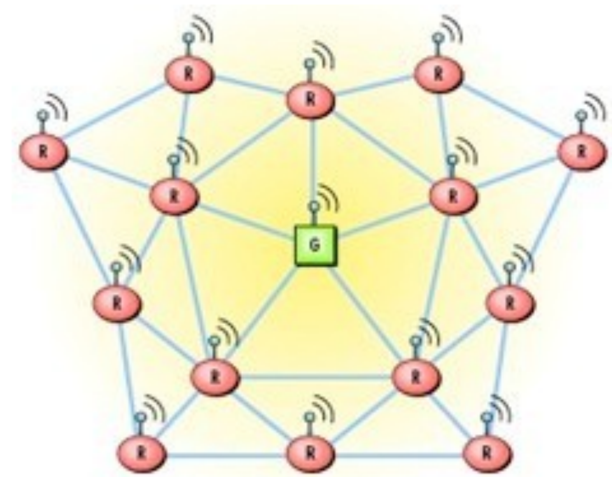
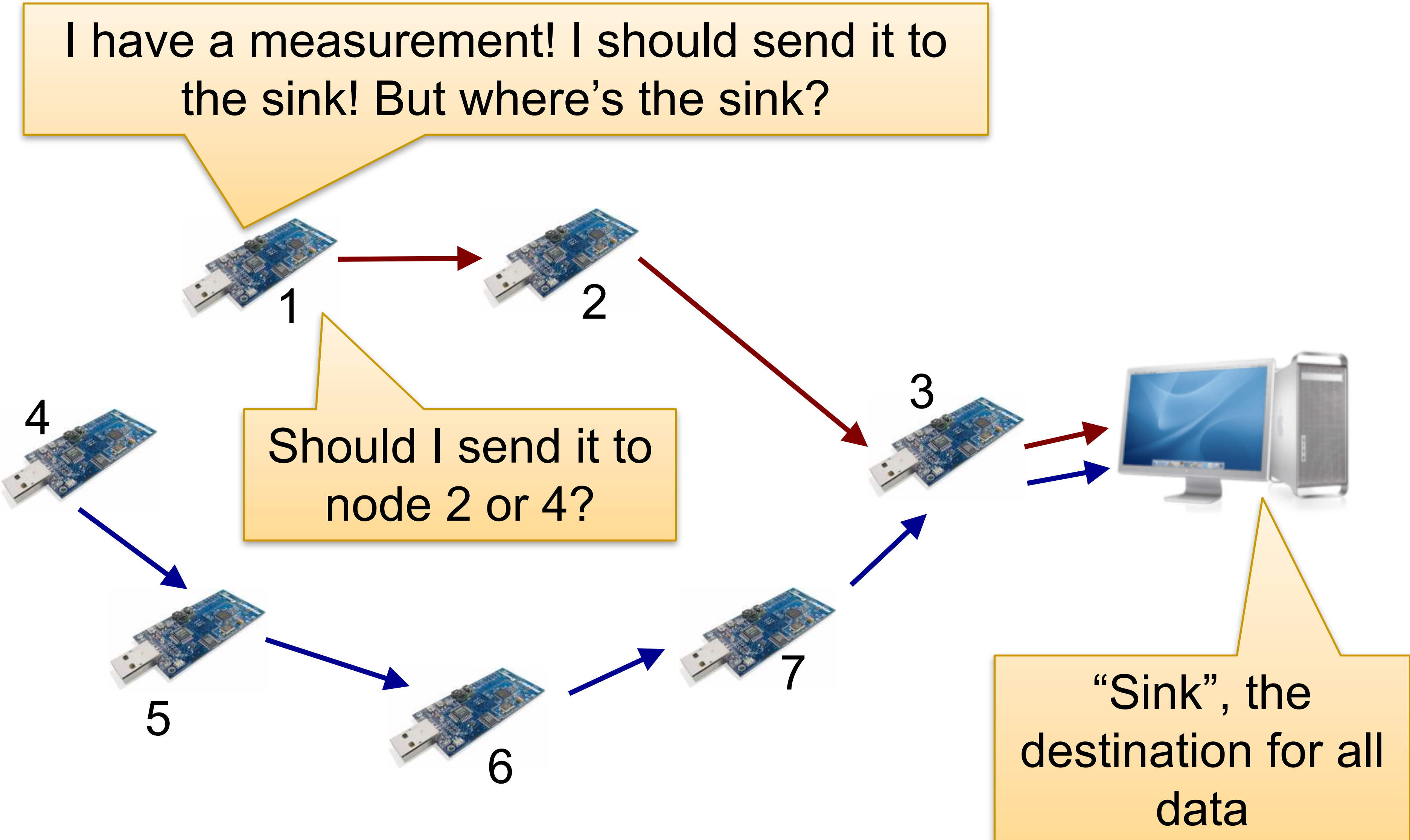
Data collection with many-to-one routing



Timofei Istomin
timofei.istomin@unitn.it
Rajeev Piyare
piyare@fbk.eu

Multi-hop data collection



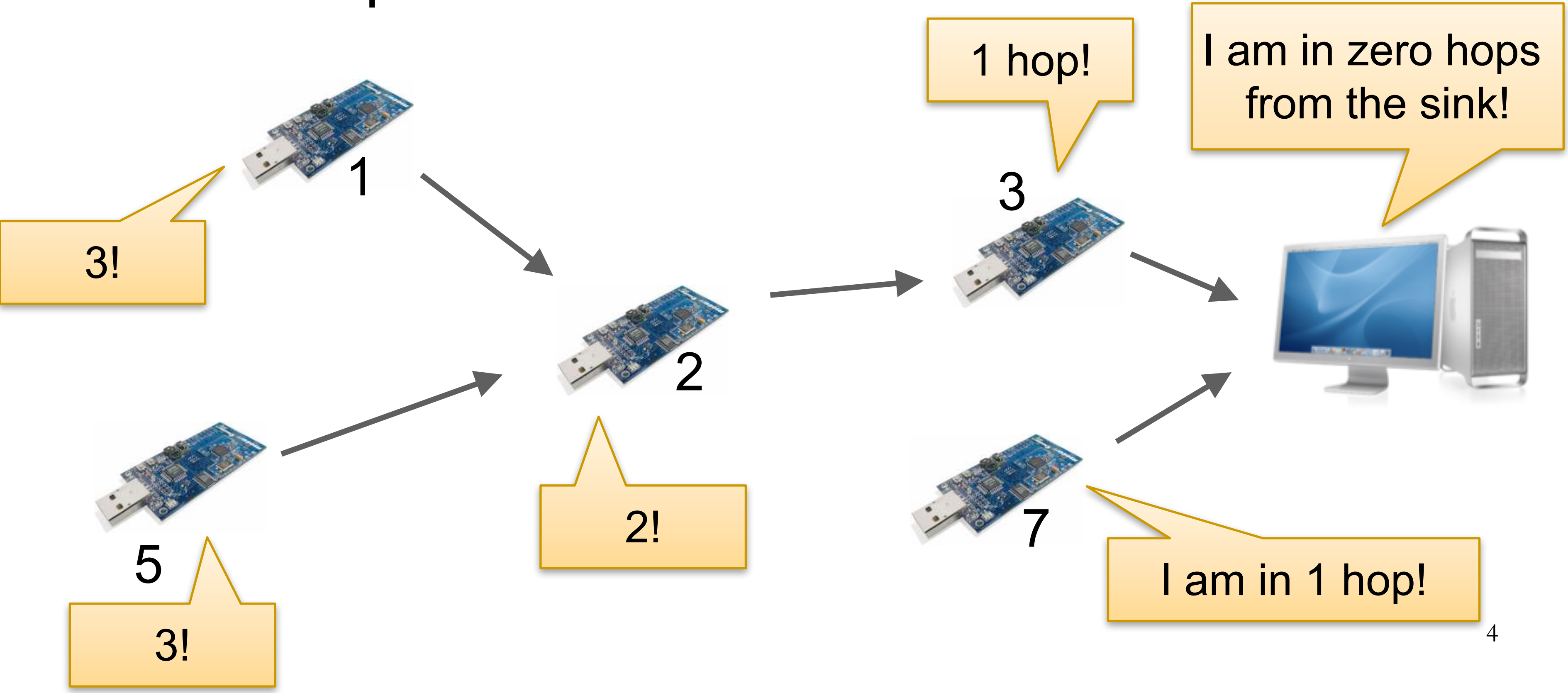
Many-to-one routing

- **Routing** — building routes (multi-hop delivery chains) from sources to destinations
- **Many-to-one routing** — from many sources (all sensors) to just one destination (the sink)
  - It is used for data collection in WSNs

- How to build these routes?
  - We will construct a minimal-cost spanning tree
  - the path cost (the *routing metric*) is a value reflecting the effort required to deliver packets along the path

Hop count

- The simplest routing metric: *hop count*
  - The path cost = its length (hop count)
  - The longer the path, the higher the cost
- It is simple but bad for wireless



Building the tree (routing)

- The sink is the root of the tree, it initiates the process by broadcasting a *beacon* with *h=0*

- When a non-root node receives a beacon with a metric *h* that is better than its current one, it
  - considers the source of the beacon as the *parent*
  - sets its own metric to *h+1*
  - broadcasts a beacon with *h+1* as the path metric

Data forwarding

- After the routes have been constructed, the nodes can send data towards the sink (by sending them to their parents)

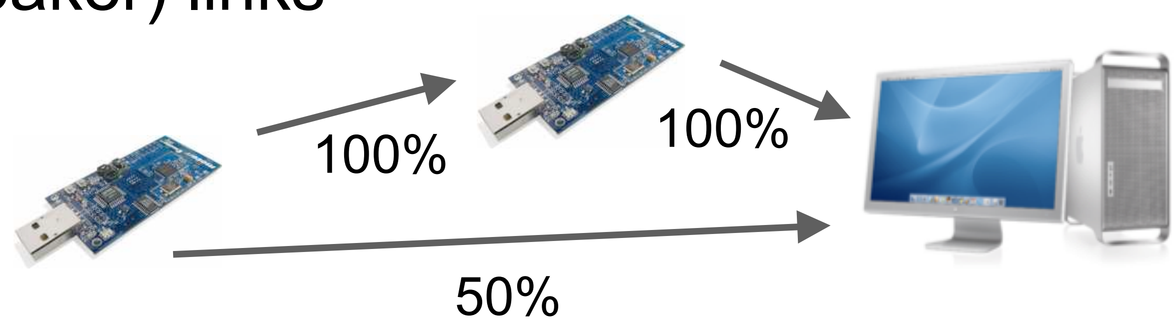
- When a node receives a data packet, it should forward (relay) it to the parent

Periodic tree updates

- The protocol should adapt to changes
  - Environment is dynamic, nodes may fail or new nodes may be installed, so
  - the build-once-forever approach will not work
- From time to time the routes should be refreshed
- One way of doing it: rebuild the tree from scratch
  - The root periodically sends a new beacon
  - When a node receives a **new** beacon, it accepts the new metric without checking it against the old one
  - How to tell a new beacon from an old one? Use sequence numbers!

Link metrics

- Why is the hop count metric bad?
  - the nodes will try to minimize the number of hops in the path, choosing longer (and therefore weaker) links



- Better would be to measure the reliability of all links and consider it when computing the path costs (not that easy)

Link metrics

- Provided by the radio transceiver
  - RSSI = received signal strength indicator
  - CORR = received signal "quality", correlation of the received signal with the one sent. Called LQI in TinyOS.
- Estimated by software
  - ETX = Expected transmission count, how many times on average it is required to transmit a packet to have it delivered to the receiver
    - requires sending many packets along the link and collecting statistics

Which to use?

- ETX is quite difficult to implement
- CORR/LQI is not modeled in Cooja
- RSSI is not reliable, and it is difficult to convert to a reasonable cost value

- Use the hop count, but filter out too bad links, based on a RSSI threshold
  - E.g. consider everything below –95 dBm inacceptable (just ignore such beacons)

Reading the HW link metrics

```
components CC2420PacketC;
TreeRoutingP.CC2420Packet -> CC2420PacketC;


int rssi =
      (int8_t)call CC2420Packet.getRssi(msg) - 45;


/* You can also use LQI on real hardware */
uint8_t lqi = call CC2420Packet.getLqi(msg);
```

use exactly this conversion!

Use it inside the `Receive.receive()` event to get the metrics for the received packet

Exercise: data collection

- Implement a NesC component that
  - constructs many-to-one routes
  - provides a service to deliver data to the sink
- Start from *routing*, when it works implement *forwarding*
- You have 2.5 labs to complete this exercise
- **Keep it simple**, make it work, then complicate (if you want)
- Later we'll compare the performance of your protocol with the de-facto standard CTP

Program structure

- Use the provided application modules
  - `AppP.nc, AppC.nc, MyCollection.{h,nc}`
  - wire them to *your* data collection component `MyCollectionC`

Components

- MyApp
  - uses interface MyCollection
  - Initiates the construction of the collection tree and after some time starts sending data to the sink periodically
- MyCollectionP (to be implemented)
  - provides interface MyCollection
  - initializes the radio on boot
  - implements routing (construction of the collection tree) and data forwarding

The collection interface

```
#include <AM.h>
interface MyCollection {
  command void buildTree();

  command void send(MyData* d);

  event void receive(am_addr_t from,
                     MyData* d);
}
```

to be called on the Sink

to send data to the Sink

to receive data on the Sink

```
typedef nx_struct {
  nx_uint16_t seqn;
} MyData;
```

Data packets

- Use this `nx_struct` for the data packets
  - it should contain our application-level structure `MyData` inside

```
typedef nx_struct {
  nx_uint16_t from;
  nx_uint16_t hops; // optional
  MyData data;
} CollectionDataPacket;
```

Implementing the routing

- Implement the interface for building the tree (used on the sink only)
  - **`command void`** `buildTree();`
  - when it is called, start broadcasting beacons periodically
- Use the `nx_struct CollectionBeacon`
- Wire to an `AMSender` and `AMReceiver`
- When you `receive` a beacon, update your metric and the parent, and rebroadcast the beacon (if appropriate)

Tips

- Use a (strictly) monotonous routing metric
  - otherwise routing loops will appear
  - hop count is such
- Explicitly handle the overflow of the path metric and sequence numbers!
- Use random delays to desynchronize the beacon broadcasts

```
component RandomC;
uses interface Random;
call Random.random16();
```

Setting up the simulation

- In Cooja, create a simulation with UDGM radio model with constant loss
- Create a small multi-hop network (3-5 nodes)

- When everything works, switch to a lossy model (UDGM distance loss or MRM)
- Try with larger networks

Implementing the forwarding

- Wire to a new pair of `AMSender` and `AMReceiver` instances
  - So you'll have separate send/receive functions for beacons and data packets
- Create a separate `message_t` for data packets, so you'll have two of them:
  - `message_t beacon_output;`
  - `message_t data_output;`
- Set the number of retries for `data_output` every time you send a data packet

Test settings

- Set the tree refresh rate to 2 minutes
  - `#define REBUILD_PERIOD (120*1024L)`
- Set the number of retries to 3
- Use the provided Makefile and Cooja script
  - don’t change anything in the Cooja script you use for the competition ;)