In Theoretical Computer Science, researchers tend to design an algorithm in an abstract way.

An algorithm should be able to be used in many different situations.

It is up to the developer to decide the way it should be turned into code for a real system.

Going from theory into practice is hard – requires programming skills in addition to knowledge in algorithm theory.

The developer also finds many limitations due to the given hardware and software specifications.

In WSN this is further augmented due to the extremely limited resources and also due to the heterogeneous nature (both in terms of hardware and software).
Implementing Algorithms for Wireless Sensor Networks

Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Networks

- **Algorithm library** for heterogeneous sensor networks
- **Easy integration** into available systems
  - Sensor Node OSs such as iSense, Contiki, TinyOS
  - Simulation environments, for example Shawn
- **Hide complexity** of underlying architecture
- **Flexible** architecture for algorithm development
- Algorithms are **interchangeable**
- Possible to create **layered structure** of algorithms
- Allow for **cross-layer design**
Basic Design Idea

- Concept for Radio
  - Contiki Radio
  - iSense Radio

- Concept for Routing
  - template <typename Radio, ...>
  - MyRouting {
    ...
  };

Supported Platforms

- WP2 OSA
- Contiki
- TinyOS
- iSense
- ScatterWeb2
- Shawn
- Linux
- Feuerware
- TriSOS
- iOS
- Android

(= fully supported; o = works / proof of concept, • = with extension)
Radio Facet

Concept

```cpp
1 concept RadioFacet {
2  // ...
3 4 typedef ... block_data_t;
5 typedef ... size_t;
6 typedef ... message_id_t;
7 enum SpecialNodeIds {
8  BROADCASTADDRESS = ..., 9 NULL_NODE_ID = ...
10 11 };
12 enum Restrictions {
13  MAXMESSAGE_LENGTH = ...
14 15};
16 int enable_radio();
17 int disable_radio();
18 int send( 19 node_id_t receiver,
20 size_t len, block_data_t *data
21 );
22 node_id_t id();
23 int reg_recv_callback(...);
24 int unreg_recv_callback(int idx)
25 }
```

Usage example

```cpp
1 class MyApp {
2 void init(Os::AppMainParameter& amp) {
3   radio = &wiselib::FacetProvider<...>(amp);
4   radio->enable_radio();
5   radio->reg_recv_callback<MyApp, &MyApp::recv>(this);
6 }
7 char msg = "Hello World";
8 radio->send( 9 Os::Radio::BROADCASTADDRESS,
10 size_t(len), (Os::Radio::block_data_t *)[m]
11 );
12 }
13 }
14 void recv(Os::Radio::node_id_t from,
15 Os::Radio::size_t size
16 Os::Radio::block_data_t *buf) {
17 debug->debug("got %s from %d", from,
18 buf);
19 }
20 }
```

Problems when Sending Messages

- How to pass a message when network is heterogeneous?
  - Different bit widths
  - Alignment issues
  - Byte order
- How to identify own messages?
  - Radio may be used by multiple algorithms
- How to flexibly register a callback?
  - No virtual inheritance
  - No C function pointers
Problems when Sending Messages

- How to pass a message when network is heterogeneous?
  - Different bit widths
  - Alignment issues
  - Byte order
- How to identify own messages?
  - Radio may be used by multiple algorithms
- How to flexibly register a callback?
  - No virtual inheritance
  - No C function pointers

The Radio Facet

Problems when Sending Messages

- How to pass a message when network is heterogeneous?
  - Different bit widths
  - Alignment issues
  - Byte order
- How to identify own messages?
  - Radio may be used by multiple algorithms
- How to flexibly register a callback?
  - No virtual inheritance
  - No C function pointers

Heterogeneity

Bit Width

<table>
<thead>
<tr>
<th>Write int</th>
<th>Jennic</th>
<th>MSP430</th>
</tr>
</thead>
</table>

Alignment

<table>
<thead>
<tr>
<th>uint16_t at odd address</th>
<th>Shawn (Desktop)</th>
<th>MSP430</th>
</tr>
</thead>
</table>

Byte Order

<table>
<thead>
<tr>
<th>Write an uint32_t</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D</td>
<td>D C B A</td>
<td></td>
</tr>
</tbody>
</table>

The “Double Problem”

- IEEE Standard for Floating-Point Arithmetic (IEEE 754)
  - Single precision: 4 bytes
  - Double precision: 8 bytes
- E.g., msp430-g++ (default)
  - Single precision: 4 bytes
  - Double precision: 4 bytes
**Solution**

- Make use of **fixed size** data types
  - Include header stdint.h
  - Use data types uint16_t, int32_t, ...
- Provide “clever” read/write methods
  - Take care of platform differences
  - Do the right thing for all datatype/platform combinations
- **Template specialization**
  - Only needed conversions will be compiled
  - Easy to add new conversion rules for new platforms/datatypes

⇒ Developer does not have to worry about platform details!

**Serialized Serialization provided by the Wiselib**

```cpp
1 template<typename OsModel_P, typename BlockData_P, typename Type_P>
2 inline Type_P read(BlockData_P &target) {
3     return Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>:
4         ::read(target);
5     }
6 }
7
template<typename OsModel_P, typename BlockData_P, typename Type_P>
8 inline void read(BlockData_P &target, Type_P& value) {
9     value = Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>:
10         ::read(target);
11 }

12 template<typename OsModel_P, typename BlockData_P, typename Type_P>
13 inline typename OsModel_P::size_t write(BlockData_P &target, Type_P& value) {
14     return Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>:
15         ::write(target, value);
16 }

- **Basic functions for read() and write()**
- Use Serialization class: Passing OS, endianness, block data, type
- **Template specialization:** Automatically generate platform dependent code

**Example: Generic Implementation and Specialization**

- **Generic implementation:** Used by default

```cpp
1 template<typename OsModel_P, Endianness, typename BlockData_P, typename Type_P>
2 struct Serialization {
3     8 static inline size_t write(BlockData_P &target, Type_P& value) {
4         for (unsigned int i = 0; i < sizeof(Type); i++)
5             target[i] = s((BlockData_P::value + i));
6     }
7     0         return sizeof(Type);
8 }

9 template<typename OsModel_P, typename BlockData_P, typename Type_P>
10 struct Serialization<OsModel_P, WISELIB_BIG_ENDIAN, BlockData_P, Type_P> {
11     4     static inline size_t write(BlockData_P &target, Type_P& value) {
12         for (unsigned int i = 0; i < sizeof(Type); i++)
13             target[i] = s((BlockData_P::value + i));
14     }
15     0         return sizeof(Type);
16 }
```

- **Read/write care for heterogeneity**
- **Template specialization for each specific platforms (possible)**
- Where not specialized, use default implementation

**Example:**

- Make use of `fixed size` data types
  - Include header `stdint.h`
  - Use data types `uint16_t`, `int32_t`, ...
- Provide “clever” read/write methods
  - Take care of platform differences
  - Do the right thing for all datatype/platform combinations
- **Template specialization**
  - Only needed conversions will be compiled
  - Easy to add new conversion rules for new platforms/datatypes

⇒ Developer does not have to worry about platform details!
### Example: Floating Point Specialization

- Template specialization for `double` values
  ```cpp
  template< typename OsModel_P, typename BlockData_P>
  struct Serialization<OsModel_P, WISELIB_LITTLE_ENDIAN, BlockData_P,
      double>
  {
    public:
    static inline double read( BlockData* target )
    {
      return FpSerialization<OsModel, WISELIB_LITTLE_ENDIAN, BlockData,
          double>::read( target );
    }
    static inline double write( BlockData* target, double& value )
    {
      return FpSerialization<OsModel, WISELIB_LITTLE_ENDIAN, BlockData,
          double>::write( target, value );
    }
  };

- Automatically adapt to platform via `sizeof(double)`
  as template argument
- `FpSerialization`: Same principle as Serialization class

### Accessing Message IDs

- Make use of `serialization`, also in algorithms!
  ```cpp
  #include "util/serialization/simple_types.h"
  
  void
  <MyAlgorithm<OsModel_P, Radio_P, Debug_P>::
  receive( node_id_t from, size_t len, block_data_t *data )
  {
    message_id_t msg_id = read<OsModel, block_data_t, message_id_t>( data )
    if ( msg_id == MyMessageId )
    {
      ... }
  }
  ```

### Message Identification

- Very simple concept for messages:
  - **Each message** has an identifier in the first byte(s)
  - **Message id type** is defined in radio
    - **Always** use `Radio::message_id_t`
  - All radio facets are adjusted for same `message_id_t`
    - **Currently** `uint8_t`, may change to `uint16_t` soon
  - See [www.wiselib.org/wiki/design/messages/id_allocation](http://www.wiselib.org/wiki/design/messages/id_allocation)

### Generic Message Composition

- In own messages, make use of `serialization` and **type definitions**
  - Define buffer array as **only** data member
    ```cpp
    template<typename OsModel_P, typename Radio_P>
    class MyMessage {
      message_id_t msg_id();
      void set_msg_id( message_id_t id )
      { write<OsModel, block_data_t, message_id_t>( buffer, id ); }
      node_id_t source();
      void set_source( node_id_t id )
      { node_id_t source() { return read<OsModel, block_data_t, node_id_t>(); } }
      size_t buffer_size() { return size_of_message(suffix); }
      enum data_positions {
        MSG_ID_POS = 0,
        SOURCE_POS = sizeof(message_id_t),
        NEXT_POS = SOURCE_POS + sizeof(node_id_t),
        ... };
      block_data_t buffer[MAXMESSAGELENGTH];
    };
    ```

---

Ioannis Chatzigiannakis  Pervasive Systems  Lecture 22  15 / 72

---

Ioannis Chatzigiannakis  Pervasive Systems  Lecture 22  16 / 72

---

Ioannis Chatzigiannakis  Pervasive Systems  Lecture 22  17 / 72

---

Ioannis Chatzigiannakis  Pervasive Systems  Lecture 22  18 / 72
Generic Message Composition

- When sending message over radio, cast message to block data:
  ```
  radio().send(destination, message.buffer.size(), (block_data*)&message);
  ```
- On reception, cast block data to message:
  ```
  MyMessage *message = (MyMessage *)buffer;
  ```

Problems with STL

- STL uses all kinds of C++ features like...
  - `new/delete`
  - RTTI
  - Exceptions
  ⇒ Bad, some platforms do not support those!
- That’s even true for other “slim” versions like the uSTL (http://ustl.sourceforge.net)

Solution: pSTL

- “pico” version of the STL
- implements a subset of the STL
  - but usable on limited platforms
    - does not require `new/delete`, exceptions, RTTI
    - not even a dynamic memory
    - resource-efficient implementation
- replace pSTL with “normal” STL any time if you need something pSTL doesn’t provide
  → Almost full STL power even on limited nodes
  → Code can be easily ported between STL and pSTL

pSTL Design

- STL-Code works with small modifications
- But don’t use `const, new, etc...`
  (code size and portability)
- Only `++iter` supported, not `iter++`
  (easier to maintain)
- Currently only statically sized containers available (will change soon)
  (do not require dynamic memory)
- Some details (like allocator passing) are different
  (because allocation is different)
**Distributed Algorithm Engineering**

**Wiselib Implementation Insights**

**Hierarchical Network Structures**

**Evaluation**

---

**The need for Scalable Network Structures**

- Current off-the-shelf WSN technologies:
  1. allow short range message exchanges.
  2. employ flat network organization structures for message exchanges, data aggregation and actuators operation.
  3. typically allow the operation of a few dozens of nodes.
- Many of the proposed applications assume large node populations densely deployed over sizable areas.
- City Scale deployments: CitySense, SmartSantander . . .
- It is important that future WSN have scalable network structures that
  1. achieve appropriate levels of organization and integration.
  2. are achieved seamlessly and with appropriate levels of flexibility.

---

**The need for Adaptation**

- A large variety of approaches have been proposed for grouping nodes in order to achieve network scalability.
- Some have been proposed as stand alone methods, others incorporated as sub protocols in larger solutions.
- Unfortunately, none of them has been widely adopted by the community
  1. extremely few software implementations for real WSN
  2. cluster formations remain static throughout the execution of the networks
- Technology expects future WSN to be dependable and adaptive to:
  1. “external changes” that affect the topology of the network (e.g., due to node failures).
  2. “internal changes” requested by the application (e.g., to reduce cluster sizes).

---

**Example: MapStaticVector**

```cpp
1 #include <iostream>
2 #include "util/pstl/map_static_vector.h"
3 #include "external_interface/pc/pc_os_model.h"
4
typedef wiselib::PCOsModel Os;
6 typedef wiselib::MapStaticVector<Os, int, const char*, 5> map_t;
7
8 int main(int argc, char** argv) {
9   map_t map;
10
11   map[1] = "first";
12   map[9876] = "over 9000";
13   map[42] = "the answer";
14   map.erase(9876);
15   map[815] = "flight no."
16
17   map_t::iterator iter;
18   for (iter = map.begin(); iter != map.end(); ++iter) {
19     std::cout << iter->first << " \n";
20     std::cout << iter->second << "\n";
21   }
22
23   return 0;
24 }
```

---

### Containers
- vector_static
- list_static
- map_static_vector
- priority_queue
- queue_static
- set_static
- pair

### Algorithms
- for_each
- find / search
- min / max
- copy
- heap operations
- sorting
- etc...
Our Approach

- Instead of trying to cope with all possible types of internal or external events we follow the approach of self-organization
- We propose an self-organizing algorithm that is verified to be correct using theoretical analysis
- We implement our solution by following a component-based design.
- We totally avoid implementing our algorithm as a monolithic, stand-alone piece of code.
- We conduct a thorough evaluation using an experimental testbed environment.
- For all cases, our results indicate that our approach adapts to the external and internal changes.

Self-Organizing Algorithm Overview

- The algorithm partitions the node of the network into small clusters that are then merged to form bigger clusters and so on.
- Nodes continuously monitor the local topology.
  - If they do not detect any cluster, they take the initiative to create a new one.
  - If one or more clusters exist, they join one of these using some very simple criteria.
- The network parameter $k$ is used to control the cluster size:
  - Set by the network operator and can be modified during the execution of the protocol.
  - The protocol adapts by adjusting the cluster size so that they have a diameter of $2 \times k$.
  - The adaptation to the new size requires $O(k)$ execution rounds.

Network Initialization

- Our algorithm follows the self-stabilization approach, so we do not assume any initialization phase.
- It is capable of starting from any configuration where the nodes of the network are set to any arbitrary state.
  - some nodes may consider themselves as cluster heads,
  - others may consider as members of non-existing clusters, etc..
- Regardless of this initial arbitrary state, within a bounded number of steps, our algorithms converges to a stable configuration
  - i.e., a configuration where all nodes of the network participate in a valid cluster of $k$ – hop diameter
- This is done regardless of the way that the devices are positioned in the network area.

Self-Organizing Neighborhood Discovery

- An important aspect is the ability to detect the current topology of the network.
- Simple approach: each node periodically broadcast beacon messages that include its unique id.
- Problem: Communication is carried out via a wireless channel – its quality varies over time.
- Solution:
  - Take into account the Link Quality Indicators (LQI) provided by the MAC layer for each received message beacon
    - Consider beacon messages with LQI above a certain threshold.
    - Drop messages below another LQI threshold.
  - Allow a node to miss a number of beacons within a given period of time before removing it (called the timeout period)
Self-Organizing Algorithm

Self-Organizing Leader Election

- Each node $u$ maintains an internal list with all the leader nodes that are within $k - hop$ distance.
- The list is continuously broadcast to all neighboring nodes.
- A node $u$ that has an empty list nominates itself as a local leader and inserts $\{id_u, dist_u = 0, null\}$.
- When a node $v$ that receives a list from a neighboring node $u$:
  1. for each entry $\{id_u, dist_u = 0, null\}$ it adds $\{id_u, dist_u = 1, v\}$
  2. for each entry $\{id_x, dist_x, u\}$ it adds $\{id_x, dist_x + 1, v\}$
  3. It drops duplication entries.
  4. It merges entries with the same id using the entry with the minimum id and with the minimal dist.
  5. Deletes entries with $dist > k$.

Self-Organizing Grouping (1)

- As soon as a node nominates itself as a local leader it enters a waiting period of $O(k)$ period of time.
- It waits for the self-stabilizing update algorithm to collect the other identifiers and notify for the leader identity all nearby nodes within at most $O(k)$ rounds.
- If there does not exist a node $u$ with distance less than $k$ from $v$, with lower id than $v$, then $v$ is a stable leader and initiates the cluster construction phase.
- If another node $u$ is identified (with lower id) then $v$ exits the waiting period and becomes passive.

Self-Organizing Grouping (2)

- Next, each active local leader starts a breadth-first search to identify all nearby nodes and invite them in its cluster.
- Nodes receiving the search message of local leader $u$ respond by joining the cluster of the leader.
- Since each node $v$ may follow a different local leader in its neighborhood, if $v$ decides to join the cluster formed by node $u$ it sends back to $u$ a response message.
- This process requires an additional $O(k)$ rounds.

The algorithm is self-organizing and the convergence time is $O(k)$ rounds.

Basic components and relation with Wiselib

- Cluster-head Decision (CHD). Responsible for the leader election (and re-election).
- Join Decision (JD). Methodology by which nodes decide to join cluster-heads.
- Iterator (IT). Categorizing and storing information related to neighboring nodes for other algorithms to be able to use it.
Core Component

- CHD is invoked to determine if the node will become a cluster head or not.
- If the node is a cluster-head: JD sends JOINREQUEST messages to nearby nodes.
- Upon receiving a Join Request message:
  - If JD decides to join, a JOINACCEPT message is sent back, IT is notified to store the node’s Cluster-head.
  - If JD decides not to join, a JOINDENY message is sent back.
- When all nodes have been examined the membership tables are generated by the IT and the process of cluster formation completes.

Implementation Details

The Core Component is defined in Wiselib as follows:

```cpp
template<
    typename OsModel ,
    typename Radio ,
    typename Timer ,
    typename Debug ,
    typename HeadDecision ,
    typename JoinDecision ,
    typename Iterator>

class CoreComponent {
  public:
    void init ( Radio&, Timer&, Debug&, CHD&, JD&, IT& ) ;
    void enable ( void ) ;
    void disable ( void ) ;
    void set_parameters ( parameters_t* ) ;
    void find_head ( void ) ;
    template<
        typename T ,
        void ( T::* TMethod ) ( uint8_t )>
        int reg_changed_callback ( T* obj ) ;

    node_id_t parent () ;
    cluster_id_t cluster_id () ;
    bool is_cluster_head ( void ) ;
    ...
};
```

The Cluster Head Decision is defined in Wiselib as follows:

```cpp
template<
    typename Radio ,
    typename Debug ,
    typename HeadDecision ,
    typename JoinDecision ,
    typename Iterator>

class ClusterHeadDecision {
  public:
    void init ( Radio&, Debug& ) ;
    void enable ( void ) ;
    void disable ( void ) ;
    void set_parameters ( parameters_t* ) ;
    bool is_cluster_head ( void ) ;
    bool calculate_head ( ) ;
};
```

The Join Decision is defined in Wiselib as follows:

```cpp
template<
    typename Radio ,
    typename Debug ,
    typename HeadDecision ,
    typename JoinDecision ,
    typename Iterator>

class JoinDecision {
  public:
    void init ( Radio&, Debug& ) ;
    void enable ( void ) ;
    void disable ( void ) ;
    void hops ( ) ;
    void get_join_request_payload ( block_data_t* ) ;
    void get_join_accept_payload ( block_data_t* ) ;
    void get_join_deny_payload ( block_data_t* ) ;
    size_t get_payload_length ( int ) ;
    bool join ( uint8_t* , uint8_t* ) ;
};
```
Implementation Details

The Iterator is defined in Wiselib as follows:

```cpp
template<typename OsModel,
    typename Radio,
    typename Timer,
    typename Debug>
class Iterator
{
public:
    void init(Radio&, Timer&, Debug&);
    void enable(void);
    void disable(void);

    cluster_id_t cluster_id(void);
    node_id_t parent(void);
    node_id_t next_neighbor();

    template<typename T, void (T::*TMethod)(uint8_t)
    int reg_next_callback(T* obj);

private:
    vector_t cluster_neighbors;
    vector_t non_cluster_neighbors;
    node_id_t parent;
    ...
};
```

Neighbor Discovery

Low Power and Lossy AdHoc Wireless Sensor Networks

- Basic Network Operation.
- Self maintenance, self configuration.
- Base for development of new protocols and algorithms.

How ND fits in the bigger picture

- Notification mechanism to applications.
- Messaging mechanism to send messages to neighbors.
Different Approaches

- Passive Detection,
- Hierarchical,
- Turn Based,
- Beaconing

Beaconing

Devices send Beacons every time unit. Reliable but:
- energy demanding
- constant traffic

Starting Point

FixedND
- Constant Beaconing
- Add to neighborhood after $n$ beacons.
- Remove after $m$ missed beacons.

Heavily evaluated in experiments with Clustering, Tracking and Routing during the previous years.

New Neighbor Identification

Execution with 3 Devices
New Neighbor Identification  Beacons every 1sec

After 1 Second

1 2 3

hello hello

hello

hello

hello

hello

hello

After 2 Seconds

1 2 3

hello:2

hello:1,3

hello:1,3

hello:2

Why An Adaptive ND?

We propose to adapt the Beaconing rate based on the Neighborhood changes.

- no changes → relaxed discovery
- any change → increase beaconing & update information
AdaptiveND

Turned to the concept of “polite gossip” to solve our problems.

AdaptiveND

- Beacons on **variable** Intervals
- Based on the changes of the Neighborhood
- Distributed decisions
- Same Strategy for accepting and rejecting neighbors

Stability is the key

**Stability**

is a metric defined as the number of Beacons \((k)\) in agreement with the current neighborhood of the node.

- Stable devices relax Beacons.
- Unstable devices send beacons quickly to regain Stability.

Setting the Stability Threshold

Based on the setup we provide two operation modes:

- **Fixk**: All Devices use the same Stability Threshold (suitable for **mesh or fixed networks**).
- **Averagek**: Devices calculate Thresholds based on the size of their neighborhood (useful for **Random Deployments**).

Extra parameters used to refine ND

As in most cases simple beacon exchanges are not enough we introduced some extra parameters to refine results:

- LQI and RSSI for incoming beacons.
- Bidirectional link identification.
- Add local information to beacons for neighbor feedback.
Simulations are important – they suffer from imperfections:
- Artificial assumptions on radio propagation, traffic, failure
  patterns and topologies
- We decided to evaluate the performance of our algorithm in
  real hardware environment.
- However, testbeds are expensive to set up and to maintain,
  hard to reconfigure for a different experiment and usually
  feature a fixed number of nodes.
- We decided to use WISEBED\(^1\): a Pan-European network of
  wireless sensor networks
- Consists of 750+ heterogeneous sensor nodes (such as TelosB,
  Mica2, iSense or Sun Spot equipped with different sensors)

\(^1\)http://www.wisebed.eu

### Evaluation Setup

- **WSN Simulator Shawn**
  - scalability and performance
  - Network Density and Size
  - Controlled Message and Node Failures
    - www.itm.uni-luebeck.de/ShawnWiki/

- **WISEBED testbed facilities**
  - real world implications
  - Mobility and Low-Power Scenarios
  - Different locations around the E.U.
  - Federated Experiments
    - http://wisebed.eu

### Real Hardware Testbeds

- We used 3 WISEBED testbed sites: UZL, GENEVA and CTI
  - 66 iSense nodes (20 in UZL, 26 in UNIGE, 20 in CTI)
  - 30 telosB nodes (15 in UZL, 15 in CTI)

**iSense Platform**
- 16 MHz 32 bit RISC – 96K RAM/128K Flash
- C++

**TelosB Platform**
- MSP430 16 bit RISC – 10K RAM/48K Flash
- nesC
Simulations: Scalability

Up to 90% less beacons exchanged in stable environments.

Simulations: Device Failures

Significantly increased Beaconing during the Failure Period (2).

Physical Topology of Testbeds

Adaptive Neighborhood Discovery – LQI Thresholds

We examine the Average neighborhood size with different LQI thresholds.
Adaptive Neighborhood Discovery – Beacon Interval

We examine the impact of beacon interval period and the neighbor timeout period in the detection of neighboring nodes.

Real World: Mobility Experiments

- Walk path for the mobile device and positions of fixed devices.

Real World: Lifetime Experiments

- 20% Extended Lifetime
AdaptiveND offers:
- Reduced messaging rates by 90%.
- Increased network lifetime by 20%.
- Lower network traffic.

Next Steps
- Evaluate Duty Cycling Strategies. (ongoing)
- Use AdaptiveND together with other Protocols. (ongoing)
- Large Scale Federated Experiments using Wisebed. (planned)

Propagation of Events across Modules

We examine the events generated by the Clustering while reacting to events generated by the Neighborhood discovery module.

Effect of Channel Failures on Adaptation Process

- Channel failures refer to a situation where a node is unable to successfully send most of its outgoing messages due to temporary noise on the wireless communication medium.
- We emulate by using a node called “the Jammer”: continuously broadcasts big messages in order to create collisions, reduce link quality and in general reduce the message delivery rate.
- The Jammer has normal communication range, identical to all other nodes.
- We position it in such a way to disrupt almost 50% of the network.
Effect of Node Failures on Adaptation Process

- Node failures refer to a situation where a node suddenly stops communicating with its neighboring node.
- We emulate node failures by switching off the “faulty” nodes.
- We conduct experiments in which, after five minutes, the running nodes randomly disable themselves with a 50% chance.
- Then after an additional five minutes, the remaining running nodes randomly disable themselves with a 50% chance.
Total Number of Events generated

![Graph showing the total number of events generated over time for different numbers of nodes.]

- 2 Nodes
- 6 Nodes
- 10 Nodes
- 15 Nodes

---

Ioannis Chatzigiannakis
Pervasive Systems
Lecture 22  72 / 72