Embedded Network Sensor Data and Decision Management

Emmanuel Caillé, *M.Eng, DCU*

***Abstract*—As sensor networks are growing, it is more an more difficult to efficiently handle data and decision within the network. Different solutions have been studied in previous works, but none of them allow to rule-program an heterogeneous topology-independent network. A new system is created to solve this, it uses broadcast to organise the network as a tree and then send queries, the answer is built while going up the tree. A prototype sensor network has been successfully programmed with rules that use global query to get an average value from the network, some timing analysis has been done to check the scalability of the system and highlight possible improvements. This prototype is a proof that this system can efficiently handle data in large networks; and the timing analysis has shown that even better performance are possible.**

***Index Terms*—M.Eng, sensor networks, query processing, rule-based programming, data acquisition, network topology independance.**

I. Introduction

WITH the improvements in circuit manufacturing, sensors boards with processing capabilities are now cheap and powerful, allowing the fast development of sensor networks. Those networks get bigger, have more and more data and can process more and more information. This potentially huge amount of information needs to be efficiently handled, especially if the network has to be programmed to take its own decisions.

Sensor networks usually contain a lot of nodes, nodes that can be different and that can host different kinds of sensors and actuators. The network topology is not always simple and can change frequently, some nodes can become unreachable whereas some others can have several network connections. Gathering data or sending commands to a sensor network is not trivial.

Several systems exist to manage data in a sensor network or decentralised system; most of them are inspired from biology examples such as the human nervous system, the insects swarms or the cells communication. But none of them allow an efficient management of data and programming in an heterogeneous sensor network.

Manuscript last revised September 5, 2014.

How can data be efficiently aggregated in an heterogenous sensor network ? How can any node be the center of the network and receive this aggregated data ? How can a sensor network be programmed with rules using this data ?

In this project a new system is implemented using C++ and Boost [1] on BeagleBone boards. The system use tree-shaped queries to aggregate data within the network, reducing the network load and allowing the system to be topology independent. The nodes can be programmed with rules in the same way they are queried for data. Different scenarios are tested using the example of a smart building. Some performance tests are performed to asses the system performance and to know the influence of the number of nodes and their processing power.

II. Existing managment models for distributed systems

A lot of different solutions are used to manage data in distributed systems and sensor networks, all with different levels of centralisation and programming abilities.

A. Cyber-physical

The first kind of solution is the cyber-physical model, where the actions are separated on two levels : the physical-reflex space with real-time reactions but no knowledge of the full system, and the cyber space where gloabal decisions can be taken but with more latency [2]. This model is interesting for large-scale fixed systems but the cyber level requires a centralised server and is not network independant.

B. Swarm behaviour

A second solution that doesn't need a centralised server is using the swarm behaviour; the three principles (separation, alignment and cohesion [3]) allow a decentralised system to self-organize efficiently by using direct and indirect communications between nodes [4]. It's really efficient to manage moving systems and cover a large area [5], but not to transmit information from a node to an other.

C. Cell communication

An other solution, inspired from biology cell communication, consists in sending the information to every nodes, and a node will only accept it if it has the correct receptor [6]. This solution is decentralised and network-independant, it will keep working if the network is modified; the downside is that it creates a lot of traffic on the network when it could be avoided.

D. Tree-shaped network

A last solution is to send queries to the network from any node, and the query will create a tree in the network to reach every node and aggregate the answers on the way back. This has been implemented with TinyDB [7] but the team mainly focused on power consumtion and low-level operations, using a specialized operating system for nodes (TinyOS).

This project will use similar principles to TinyDB but will focus on the rule-programming aspect and the interoperability, while keeping the network independance and distribution aspect of TinyDB.

III. Building a sensor network management system

A. Global design

The system is built to follow several concepts which ensure that the result will answer the problematic:

- The system work whatever the network topology, and keep working if the topology is changed.
- The system is totally decentralised, meaning that any node can either be considered as a server or a client or a querier.
- Decisions can be taken both at the local level (on a single node) or at the global level (using information from several nodes).
- The system allow some interoperability, different kind of nodes should be able to communicate with each other.

Because of time constraint, everything can't be implemented, the focus has be given on some functionalities more than others:

- Querying a node for local sensor values
- Programming a node to act according to local values
- Querying the whole network for an aggregated value
- Programming the whole network (or a node) to act according to an aggregated value

To do this, the same software run on each node, it includes a server that listen for incoming messages, a client that can be used to send messages, and a processing part that process messages, answer them and take decisions. Each node send a broadcast heartbeat so that each node can know its neighbours. To minimise the number of transmitted messages and allow a network-independent execution, when sending a query for the whole network, the network is arranged as a tree that cover every nodes, and when answering the answers are aggregated at each branch. This is explained with further details in the next section.

B. Hardware

The system studied is a sensor network, it consists of nodes linked together by at least one connection. The nodes need a network connection, a good processing capability and the possibility to have sensors and actuators. Embedded Linux boards answer all those needs; the BeagleBone has been choosed for this project, as it is a cheap board with a relatively powerful processor and a lot of input/outputs. Each node can have a various number of sensors (such as thermometers, light sensors) and actuators (such as light, heaters, central heating).

C. Software

The system needs to be efficient, multi-platform and easily modified; it means the programming language have to be low level and compiled, multi-platform (and/or using a multi-platform library), and object-oriented. C++ seems to be the best choice, as it provides everything that is needed; the Boost cross-platform library will be used to handle the networking (`boost::asio` [8]) and threading (`boost::thread` [9]) aspect of the software.

D. Implementation

Each node runs an heartbeat server that periodically sends a UDP broadcast packet with the node identifier; every node directly connected (on the network layer 2) will receive it and add the node to its neighbour list.

Data are exchanged between node by sending serialized objects (messages) over a TCP connection. TCP has been chosen to ensure the reliability of the transmission and to ease the answering process (the connection stays opened until the answer is received).

When dealing with a global query (query for several nodes), the network [Fig.1] will be organised as a tree [Fig.2], with the querier as the root [Fig.3], and when answering the answers will be aggregated at each leaf. To do that, the querier will send the message to each neighbour, each neighbour will do the same, and so on so forth until the network is totally covered. A node

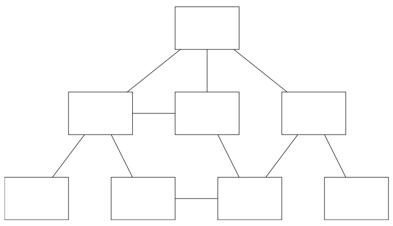

Fig. 1. A network (at level 2).

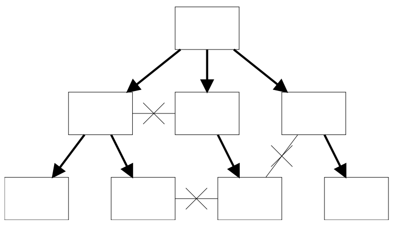

Fig. 2. Network organised as a tree, ununsed links are crossed.

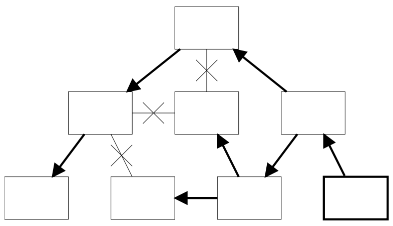

Fig. 3. Same network organised as a tree with a different root.

will wait to get an answer from all its sons to answer to its parent with the aggregated answer, it means that each node only send one answer. To avoid creating loops, each node has a list of the messages it has answered (or is answering), and if it receives it again the message is discarded.

The server and sending part is coded as a template, it can be used to send and receive any kind of messages. Two types are fully implemented : the data queries to get information from sensors, and the programming queries, to program a node with rules. There is a third type of message: the answer message that is used to answer queries with the values asked.

Sensors and actuators have both a name and a type (implemented as a string). The couple name/type should be unique, it can be used to identify a particular sensor/actuator in queries or rules. Because of time constraints, only the sensor/actuator type is used as identifier in the current implementation.

Programming a node with a rule is done by adding the rule in the node's rules list and running it in a thread. A rule consists of a test, an action and a refresh delay. The rule is run according to the delay, if the test return true, the action is excecuted. The test consists of a sensor description (like in queries), a comparison operator and a value; the test send a query (it can be a global query) and compare the result to the stored value. The action consists of a description of actuators and an output state. A rule is installed on a node only if this node has an actuator concerned by the rule.

E. Class diagram

The code uses a central class called Node, it is where all the instances are stored. It stores and runs the servers (one for data query and one for rule-programming messages), the heartbeat server, the list of sensors and actuators, the list of rules, the neighbours nodes and the list of already processed messages. See Fig.4.

F. Code

All the code has been developed specificaly for this project, except the serialization of messages which is managed by a class template from a `boost::asio` example [10]. The code listing can be found in [11].

One of the most challenging part of the project was to make sure that two different queries can be processed at the same time on a node. It is done by using threads and by sharing the `boost::asio io_service` [8], the class that manage the network connections. The service associated to this class is run several times in a Thread pool.

IV. Prototype and results

A. Mockup

The tests will be done using the example of a smart building with temperature sensors and a central heating system. The building manager should be able to program the central heating to be active if the average temperature of all the rooms in the building goes below a threshold value.

The prototype doesn't include real thermometers and heating system, the sensors and actuators are virtual. The prototype network consists of four nodes: a computer, two BeagleBone Black and a BeagleBone White, they are connected as shown in Fig.5. This network is small but complex enough to test the different network features such as answer aggregation and message forwarding.

Each node runs the same version of the code, has four temperature sensors (representing four rooms), and has a different identifier (name). One of the node (BeagleBone 1) has an actuator (central heating).

B. Use case

Different use-case have been tested to check that the functionnalities are working.

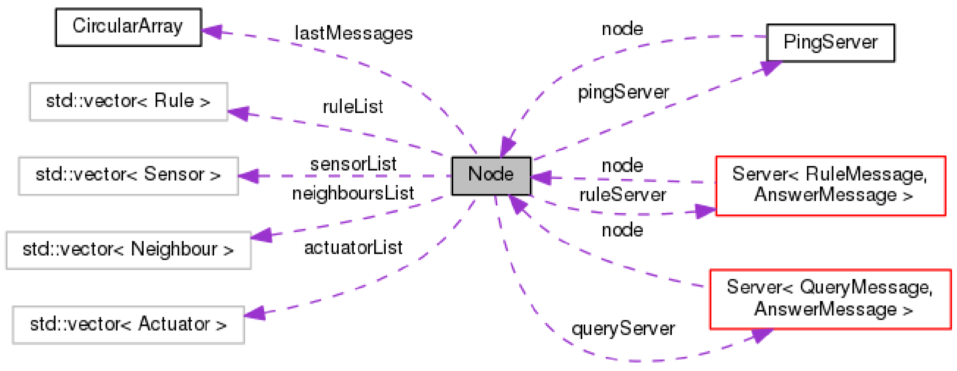

Fig. 4. The Node class and its relationships with other class.

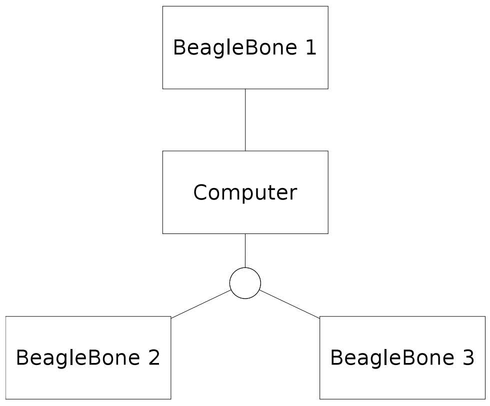

Fig. 5. The prototype network used for tests.

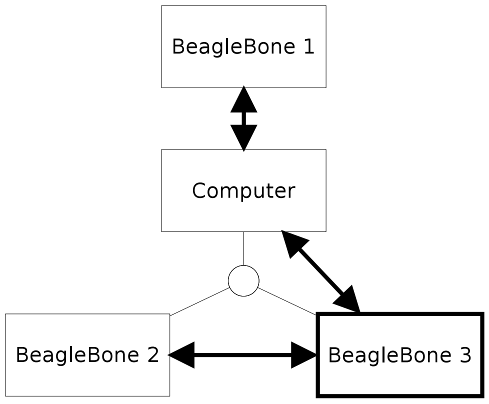

Fig. 6. Message path for a global query from BeagleBone 3.

1) Simple query: Query from BeagleBone 1 to BeagleBone 1, asking for temperature.

Query answered locally.

2) Global query: Query from BeagleBone 3 to all, asking for temperature.

The query is sent to Computer and BeagleBone 2, from Computer it is sent to Beaglebone 1.

BeagleBone 1 answer to Computer that aggregates it with its answer and sends it to BeagleBone 3.

BeagleBone 2 answers directly.

BeagleBone 3 aggregates the answers, adds its own values and gives the answer.

The tree created is illustrated on Fig.6.

3) Rule programming: Rule sent from BeagleBone 2 to all, Rule saying that, every minute, if the average value from temperature sensors is below 18 the actuator central heating is switched on.

As for the global query, the message is sent to every nodes, following the same path.

BeagleBone 1 is the only node to accept the rule, as it is the only one to have a central heating actuator.

The rule is run every minute. When run, BeagleBone 1 send a global query asking for the average temperature, the answer is used by the rule.

C. *Limitations*

Because of time constraint, some features are missing, like message priority, portability tests and user-friendly modifications.

Message priority is needed if the sensor network has to deal with time critical systems or security features. For smart building it could be smoke detectors: it is not acceptable that a smoke detection query is slowed by a temperature query.

The code is written in C++ and use Boost, so it can theoretically run on any platform with a C++ compiler and a Boost library. The objects are serialized by `boost::serialization` [12], so it should be the same for any systems, allowing two different platforms to communicate. Nevertheless, this hasn't been tested.

Rules have been designed so that it will be easier to create and implement new rules, but it still implies to modify the code and re-compile the program, it could be interesting to use something like parsing to allow the user to create rules from configuration files without modifying the code.

V. Performance and Time

The query system used (creating the tree-shape network) may induce some delays in the answering process, some additional tests have been conducted to get an estimation of those delays.

In this benchmark, one node try to send a thousand queries with different parameters, the execution time is measured and used to calculate the message rate.

Before analysing the results, it should be kept in mind that BeagleBone 2 is a BeagleBone White and is less powerful than the Black version; Computer has much more powerful processing capabilities than the BeagleBone.

A. Sequential queries performance

The first scenario is used to measure the maximum message throughput for queries to a single node with two different ways. The first way is to send the messages one by one and wait for the answer to send the next one, the second is to send all the messages in the same time, without waiting for the answer (threaded queries). To increase the number of forwardings, the link between BeagleBone 3 and Computer is cut for the two last tests. The results are shown in Table I and Fig.7.

TABLE I
One-node query rate per second

		Sequential	Parallel
1	BB 3 ↔ Computer	140	170
2	BB 3 ↔ BB 2	125	110
3	BB 3 ↔ Computer ↔ BB 1	100	125
4	BB 3 ↔ BB 2 ↔ Computer	59	62
5	BB 3 ↔ BB 2 ↔ Computer ↔ BB 1	42	42

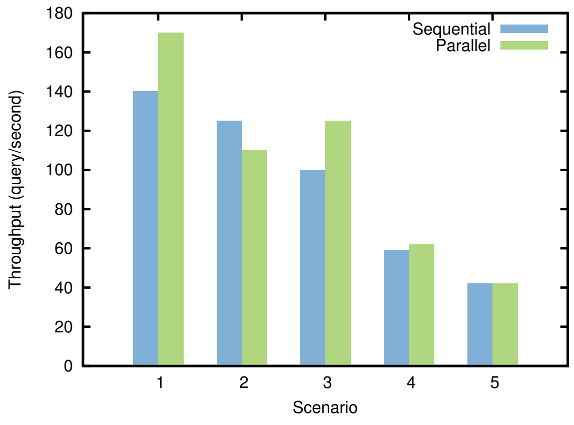

Fig. 7. Query rate for different one-node query scenario.

Those results show two things. The first is what was expected, adding more nodes as intermediary reduce the throughput, and it seems that the added delay is not linear. The second interesting thing is the influence of processing power: when adding Computer as intermediate node, the throughput doesn't change much, but when adding the BeagleBone White (BB.2) it drops. And sending parallel messages doesn't improve it if there is a slow node in the way.

So parallelizing is not really efficient for single-node queries, but what about global queries ?

B. Global queries performance

For global queries, using sequential or parallel queries has an impact on the shape of the resulting tree in the network, the same conditions than for Fig.6 will give a tree as shown on Fig.8. A global query is sent from BeagleBone 3, the maximum throughput is measured and shown on Table II and Fig.9.

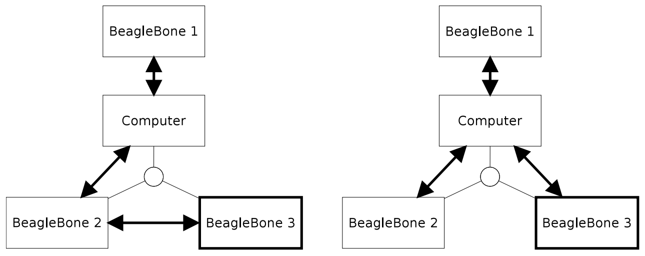

Fig. 8. Possibles message paths with a sequential sending.

TABLE II
Global query rate per second

		Sequential	Parallel
1	Computer ↔ All	32	44
2	BeagleBone 3 ↔ All	29	36

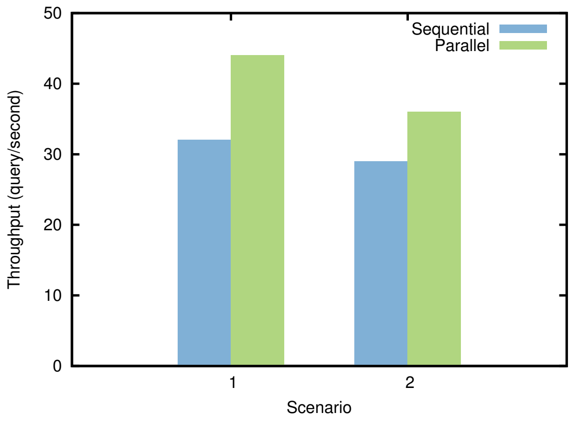

Fig. 9. Query rate for different global query scenario.

The results show that, for global queries, parallelizing is efficient. It's understandable as in certain situations it allows two node to process their answer in the same time. It would probably be even more efficient on a bigger network with several subnets.

C. Sequence diagram

The sequence diagram on Fig.10 gives more details on the time spent by a global query with the sequential sending. As for this prototype the network link are either Ethernet or USB the network delay is really low. Most of the time is spent on the nodes, especially the slow nodes like BeagleBone 2. On of the reasons of this delay is that nodes try to send the query to all their neighbours, including the ones that already received it. This could be improved by future work on this project.

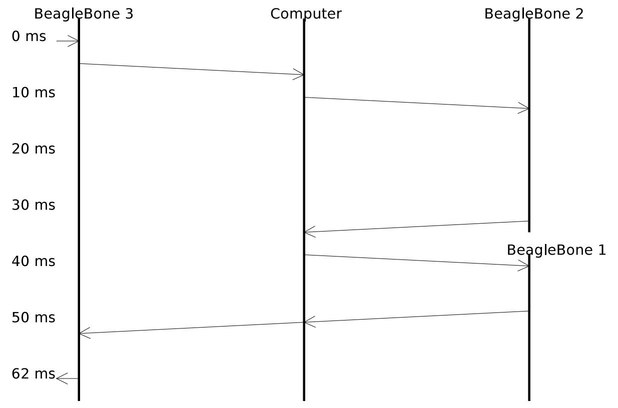

Fig. 10. Sequence diagram of a global query with sequential sending.

VI. Conclusion

This project has shown that it is possible to aggregate data in a sensor network by arranging it as a tree, and that it works with any node being the querier. This network can also be programmed with rules that use this kind of queries to take decisions. This system is efficient to reduce network load but can induce delays if the nodes have a low power processing.

In a network where every node is in the same L2 network (meaning that every node is a neighbour of every other node) the implemented solution may be inefficient for global queries. The cause is that each node will forward the queries to every other node, and even if most of times the query will be rejected because already answered, it will take time and network resources. This could be solved by using a routing algorithm that will help building the tree before sending the query. It could also be reduced by adding the list of nodes that have answered in the answer message.

References

[1] (2014, Sep.) Boost c++ libraries. [Online]. Available: http://www.boost.org/

[2] Y. Wang, G. Tan, Y. Wang, and Y. Yin, "Perceptual control architecture for cyber-physical systems in traffic incident management," *Journal of Systems Architecture*, 2012.

[3] B. A. Kadrovach and G. B. Lamont, "A particle swarm model for swarm-based networked sensor systems," in *Proceedings of the 2002 ACM symposium on Applied computing*. ACM, 2002, pp. 918–924.

[4] D. J. Stilwell and B. E. Bishop, "A framework for decentralized control of autonomous vehicles," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 3. IEEE, 2000, pp. 2358–2363.

[5] X. Cui, T. Hardin, R. Ragade, and A. Elmaghraby, "A swarm-based fuzzy logic control mobile sensor network for hazardous contaminants localization," in *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*. IEEE, 2004, pp. 194–203.

[6] F. Dressler, I. Dietrich, R. German, and B. Krüger, "A rule-based system for programming self-organized sensor and actor networks," *Computer Networks*, vol. 53, no. 10, pp. 1737–1750, 2009.

[7] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[8] C. M. Kohlhoff. (2014, Jul.) Boost::asio c++ library. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_asio.html

[9] A. Williams and V. J. B. Escriba. (2014, Jul.) Boost::thread c++ library. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/thread.html

[10] C. M. Kohlhoff. (2014, Jun.) Boost::asio object serialization on top of a socket – class template. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_asio/example/cpp03/serialization/connection.hpp

[11] E. Caillé, "Source code listing," *Appendix C – MEng Final Portfolio*, Sep. 2014.

[12] R. Ramey. (2004, Nov.) Boost::serialization c++ library. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/libs/serialization/doc/index.html