In-network Processing and Collective Operations using the COCOS-Framework

Maik Krüger, Reinhardt Karnapke, Jörg Nolte
BTU Cottbus
Distributed Systems/Operating Systems group
maik-krueger@gmx.de,{karnapke, jon}@informatik.tu-cottbus.de

Abstract

COCOS *(COordinated COmmunicating Sensors)*[1] *is a lean middleware platform for wireless sensor networks. The major programming abstractions of* COCOS *are distributed sensor spaces. All objects in such a space can be collectively addressed. This way, high-level data-parallel programming concepts such as global reductions are possible. This paper introduces the spaces of* COCOS *and describes their usage.*

1. Introduction

In the near future we will be surrounded by sensor networks to monitor the environment, bodily functions of humans or animals as well as technical systems such as cars, helicopters or planes [1]. Sensor networks will be typically composed of tiny computers with some sensing as well as wireless communication capabilities. These networks can be fairly large, ranging from a few dozens up to myriads [3] of nodes as visions of smart dust and paint already suggested. Swarms of small robots might patrol chemical plants to detect potential problems and *react* on environmental hazards e.g. by collectively applying suitable chemical agents to neutralize spilled acid.

Given the rather complex tasks we expect sensor networks to perform in the future the programming of such networks is a major challenge. Even though realistic sensor networks are currently limited to a couple of hundreds nodes at most, it is simply impossible to analyze all data outside the network due to severe bandwidth limitations. Therefore programming such networks remains a tedious task that requires parallel and distributed in-network processing with severely resource constrained computers and low-end error prone wireless networks.

The nature of sensor networks being collections of computational nodes of the same kind strongly implies a data-parallel programming approach. The sensor network then appears to the programmer as a collection of language level objects that can be grouped according to some application specific criteria such as the location of specific sensor objects or even specific sensor readings. In COCOS, all objects in such a user defined group can be addressed collectively and thus often recurring tasks on the same group of sensors or actuators can easily be expressed and implemented efficiently. Furthermore, the implementation of application specific shared data spaces that e.g. cache or replicate sensor readings to alleviate the energy consuming communication overhead is also possible with reasonable programming effort. Therefore, a general purpose data-parallel programming platform has strong benefits for both application and system programmers.

This paper is structured as follows: First we discuss the rationale of COCOS and introduce the concept of sensor spaces for parallel programming in wireless sensor networks. Section 3 describes the architecture of COCOS and some essential implementation details. Section 4 provides the results of our experiments with radio equipped Lego RCX robots while section 5 discusses related work. We finish with a conclusion and future work in section 6.

2. Sensor Spaces

Generally, each sensor node just provides a single pixel (a local sensor reading) of the whole picture. The detection of complex phenomena such as the edge of a fire requires at least some regional and sometimes global analysis of sensor data in space and time. Usually, only those nodes that detected a local phenomenon are of interest and an effective way to address only these nodes when performing a distributed and parallel analysis of sensor readings is needed. For instance, when the average temperature of all nodes with a temperature reading above a certain threshold should be determined, we would like to group those nodes together, perform parallel aggregations (global reductions) of all temperature readings periodically and transmit the aggregated result to the outside. COCOS provides sensor spaces as a basic programming abstraction to support such computational patterns. First, all nodes that detected the requested phenomenon simply join a sensor space, in this example it would contain only hot nodes. Any node may now act as an evaluator and

[1]The COCOS Project is supported by the German Research Foundation (DFG) in the SPP 1140.

perform parallel method calls on all objects within that distributed space. Thus, the space essentially acts as a distributed container for objects that can now be addressed in a data-parallel fashion. All sensor nodes are represented by language level objects that are instances of arbitrary C++-classes. Furthermore each real sensor node might register the same sensor object in multiple sensor spaces, where each space reflects a certain scope of interest.

Provided the detected phenomena are rather stable over time, we can construct suitable overlay networks that effectively reach all nodes of interest. The same holds of course for nodes that are grouped together according to static criteria such as geographic location (e.g. all nodes in a defined distance around a certain location) or according to functionality (e.g. the group of all regional managers).

However, the situation changes drastically, when either the sensor nodes are mobile, or both the wireless network as well as the detected phenomena are not stable over time. In those cases it is not possible to maintain overlay networks effectively and suitable flooding techniques have to be applied instead. COCOS reflects these differences by providing two categories of sensor spaces.

The `AnchoredSpace`, the `ConnectedSpace` as well as the `RobustAnchoredSpace` follow a topology based approach while the `ATBFloodingSpace` relies entirely on dynamic flooding techniques.

2.1. The AnchoredSpace

The `AnchoredSpace(AS)` is the simplest space implemented. It has an anchor or root node from which all group operations are distributed. When the `AS` is initiated the root node creates a tree based object space within a specified number of hops around itself. Once a routing tree has been created all messages are distributed along this tree. The `AS` is designed for static networks and expects that a node which is reached at the time of initialization can be reached at any time later on same way. If the `AS` does not meet robustness criteria for a certain application, a transport layer can be used to supplement it, or another space like the `RobustAnchoredSpace` can be chosen.

Figure 1 shows the construction of the routing tree for an `AS` for five nodes. The graphics are taken from a simulation run with 5 nodes and using a collision free TDMA MAC. In (a) only the layout is shown and the node numbers are given. In (b) node 1 starts to build an `AS` with a distribution range of 2 hops. All direct neighbors (2,3,4) receive the message and accept node 1 as their parent, which is shown by the thin lines in (c). As the number of hops is bigger than 1, node 2 redistributes the message as seen in (d). Nodes 1,4 and 5 receive the message, but 1 and 4 ignore it as 1 is the anchor and 4 already has a parent. Node 5 accepts node 2 as parent in (e). In (f) and (g) nodes 3 and 4 retransmit the message, but no new node hears them. Node 5 does not retransmit the message, because the maximum hop count was reached. Now every node knows its parent but the parents do not know their children yet, which is necessary for the `AS` to work properly. The next pictures show how the nodes inform their parents of their existence. Node 5 informs node 2 in (h) and is connected in (i). The nodes 2 (j, k), 3 (l, m) and 4 (n, o) follow, until the tree is finished in (o). Now messages can be routed from the anchor to all group members and back.

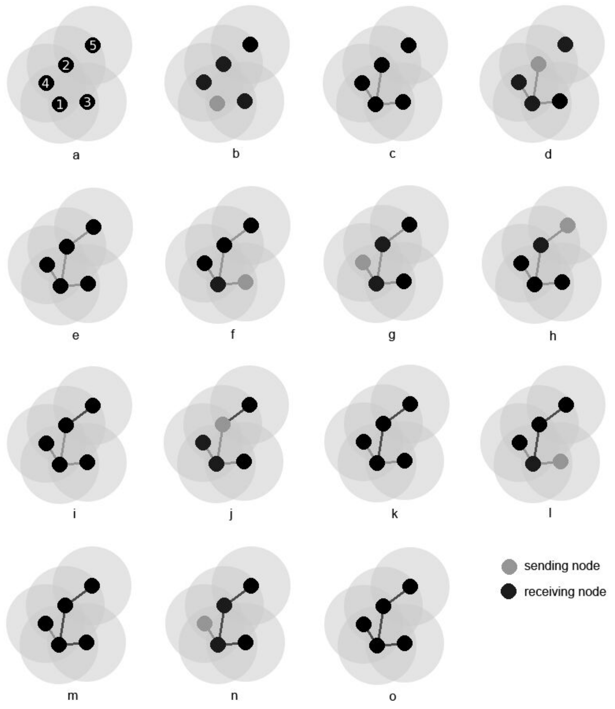

Figure 1. Building an AnchoredSpace

For the aggregation of values it is necessary to use a timeout, as messages might have been lost. The calculation of the timeouts is done in the following way. First the time needed for the communication between neighboring nodes is determined. Equation 1 shows that this $MaxReplyTimeout$ depends only on the MAC layer. The Frame size is the time which passes between a node's turn to send and the time, when it can send again. We chose to multiply this by 3 to account for the fact that both nodes may have to wait until their turn to send arrives.

$$MaxReplyTimeout = 3 * Framesize(MAC) \qquad (1)$$

When the time for neighborhood communication is known, the global timeout used at the Anchor can be determined. Equation 2 shows that the timeout for single hop communication has to be multiplied by the highest possible tree depth. This value is known, as the AS is propagated over a maximum number of hops.

$$GlobalTimeout = MaxTreeDepth * MaxReplyTimeout \qquad (2)$$

Now that the global timeout is known, every node needs to know its local timeout, namely how much time it has to send return values to its parent after receiving the message. This time is the global timeout minus the time, the message needed to get to this node and the time the message needs to get back. The time needed to reach this node is the depth of this node times half the communication single hop, as is the time needed to send an answer back. Therefore, the depth of the tree has to be multiplied with $MaxReplyTimeout$ (equation 3).

$$\begin{aligned} LocalTimeout = \\ GlobalTimeout - TreeDepth * MaxReplyTimeout \end{aligned} \tag{3}$$

Once the local timeout has passed, the node transmits the aggregated results to its parent, regardless of any still missing messages.

2.2. The ConnectedSpace

The `ConnectedSpace(CS)` is similar to the `AS`. The difference between these two spaces is that the message building the `CS` is propagated as long as the space reaches a node which wants to be a group member within a certain number of hops. The number of hops is specified at initialization. Once a routing tree is created all messages are distributed along this topology. Figure 2 shows a `CS` with a hop distance between members of 2. The node a little above and to the left from the center started building the `CS`. All its direct neighbors received the message. The node to the right wants to be a group member and thus does not decrement the hop count. Its right neighbor does not want to be a member so it decrements the counter and rebroadcasts the message. The next node wants to be a member and resets the hop counter. This continues until all messages are discarded because their hop counter reached 0. The nodes in the middle received the messages but do not participate as none of them wanted to be a member. The nodes in the lower right would like to join the group but can not because they are more than 2 hops from the last group member.

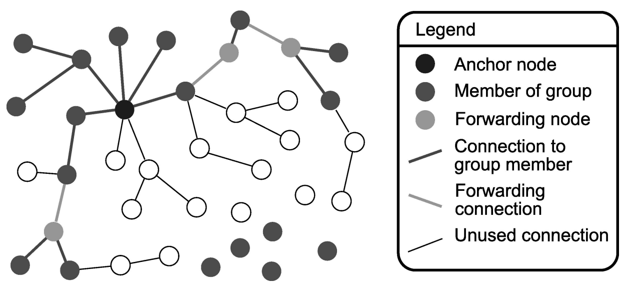

Figure 2. Building an ConnectedSpace

The timeouts for the `CS` are calculated like those for the `AS` shown in equations 1, 2 and 3. The only difference is that the tree depth has to be guessed.

2.3. The RobustAnchoredSpace

As its name suggests, the `RobustAnchored-Space(RAS)` is an extension of the `AS`. In the construction phase, nodes that received a message try to inform their parent of their existence. If the parent does not send an acknowledgement in a certain time, the message is sent again. When the parent acknowledges a child, it transmits a bit position to this child. The parent notes the position in a bit field. When results of a collective operation are collected, a parent node ticks off the received answers. If it did not receive as many answers as it has children it transmits a message containing the remaining bit field. All children check their bit position in that bit field and retransmit their results if necessary. This way, the parent node can collectively address all children, whose answers it did not receive. Those children whose bit position is not enclosed in the bit field simply ignore the message. The size of the bit field is defined at compile time and determines the number of children a node can have at most. When nodes try to connect to a parent that already has that number of children, their requests are turned down. One problem that arises when repetitions are used is that the order of messages can be affected. If the first of two messages has to be repeated three times, but the second one is sent and received without problems, the second can arrive at the destination first. To avoid this problem, the `RAS` allows only one collective operation at a time. Because of the repetitions the timeout for the `RAS` has to be calculated differently from the one of the `AS`.

The global timeout is calculated as shown in equation 4. The difference to the `AS` can be seen in the factor 2 times $MaxSendReply$ which represents the time needed if all transmissions need the maximum number of retries.

$$\begin{aligned} GlobalTimeout = \\ MaxTreeDepth * MaxReplyTimeout * \\ 2 * MaxSendReply \end{aligned} \tag{4}$$

The local timeout equals the global timeout minus the time the message needed to reach this node and the time for the way back. These are shown separately in equation 5, and calculated in the following equations.

$$\begin{aligned} LocalTimeout = GlobalTimeout \\ -TimeToThisNode - TimeForWayBack \end{aligned} \tag{5}$$

The time the message needed to reach the current node is shown in equation 6. It is the sum of the time that would be needed if no errors occurred and the time needed for all retransmission that occurred.

$$\begin{aligned} TimeToThisNode = \\ MaxReplyTimeout/2 * TreeDepth \\ +MaxReplyTimeout * TotalRetries \end{aligned} \tag{6}$$

Equation 7 shows the time reserved to transmit the message back to the anchor. It consists of the time that would be needed if no error occurred and the time it could take to send the highest possible number of retries.

$$TimeForWayBack = MaxReplyTimeout/2 * TreeDepth + MaxReplyTimeout * TreeDepth * MaxSendReply \quad (7)$$

2.4. The ATBFloodingSpace

The `ATBFloodingSpace(ATB)` is designed for mobile networks or static networks which change their logical topology, it does not have an anchor or root node, like the other spaces do. Furthermore it does not have a static topology. Therefore, no topology has to be updated and new group members can join or leave the group at any time. Every node in the space can start a group operation at every time. All group operations are distributed by flooding the message in the whole space or over a constant amount of hops around the sender. A modification which allows the distribution of operations as it is done by the `CS` is of course possible. The `ATB` does not send any explicit or implicit confirmation at message distribution, this is not necessary because the flooding of messages ensures that a message can reach each node in multiple ways. Consequently, the chance for a node to receive all messages in a dense network is high. Duplicate suppression ensures at-most-once semantics. When results are collected, the messages are sent directly (over one hop) back to the originator. Here it could be possible that messages are lost. Hence, a transport layer could again be useful.

The timeouts for the `ATB` are similar to those for the `CS`. The depth of the routing tree has to be guessed like in the `CS`, but the information whether a node is a leaf is missing. This means that an additional time has to be waited, in which leafs can detect that they are leafs by not receiving any messages from children. This time is represented in equation 8 by the last term which equals two times the time for a local communication.

$$GlobalTimeout = MaxTreeDepth * MaxReplyTimeout + 2 * MaxReplyTimeout \quad (8)$$

3. Implementation Aspects

COCOS is essentially based on a layered architecture with three layers (figure 3). REFLEX [6], the operating system we used can also be seen below COPRA, it takes care of scheduling and event handling.

In COPRA [5] (COmmunication PRocessing Architecture) different communication tasks, e.g. medium access control, are implemented in so called Protocol Processing Stages (PPSs). These PPSs can be combined to form Protocol Processing Engines (PPEs) which represent entire network stacks.

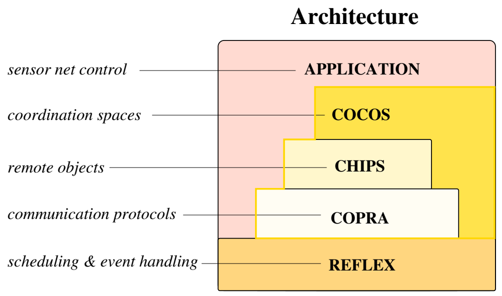

Figure 3. COCOS architecture

CHIPS (Convenient High-level Invocation Protocol Suite) enables the usage of remote method calls and uses the communication protocols provided by COPRA. This way, the remote calls can potentially be routed trough the whole network.

COCOS (COordination and COoperation Spaces) supplies a high level of abstraction for application programmers in wireless sensor networks. Using COCOS, the sensor network can be programmed as a whole, or some part of it. This is realized by extending the communication layers from COPRA and the remote calls of CHIPS to commands for logically connected groups of sensor nodes. COCOS offers an object space, where it is easy to create, use and delete object groups. An object group or object space summarizes chosen sensor nodes, so that is easy to join nodes which have special sensors or special measured values. Once an object space is created group operations on all nodes in this space can be executed. These operations can include the reading of sensor values and their aggregation as well as controlling actuators, e.g. starting a fire extinguishers in burning areas when a fire is detected.

At the moment there are three group operations implemented, the methods `apply`(), `aggregate`() and an asynchronous version of `aggregate`() which uses resynchronization. The method `apply`() is an asynchronous one way group operation, which invokes a method on all group members. The `aggregate`()-method is a synchronous group operation which returns a value that is received by aggregating all values with a specified aggregating operation. The algorithm which is used in the aggregation can be specified, by defining a class which is inherited from the abstract class `Aggregator`. The following example shows how to use the group operations.

```
class Robot {
public:
  // turn left for ms milliseconds
  void turnLeft(int ms);

  // return the current temperature
  int getTemp();
};

//creates a local Robot object
```

```
   Robot myRobot;
12
   //declares a Robot group
14 AnchoredSpace<Robot> robots;

16 SumAggregator<int> aggregator;

18 int main() {

20  // connect myRobot with group robots
    robots.join(&myRobot);
22
    // creates a 3 hop wide Robot group
24  // of type AnchoredSpace, the anchor is
        node 5
    robots.init(NodeID(5), 3);
26
    // execute the method turnLeft(3)
28  // asynchronous on all group members
    robots.apply( m2f(&Robot::turnLeft, 3) );
30
    // execute the method getTemp()
32  // synchronous on all group members
    // and aggregate the return values
34  robots.aggregate( m2f(&Robot::getTemp),
        aggregator);
    int temp = aggregator.getValue()/
        aggregator.getParticipants();
36 }
```

Lines 1-8 show the class `Robot` which represents a mobile sensor node with a temperature sensor. It supplies the methods `turnLeft`() and `getTemp`(). In lines 10-16 the needed objects of type `Robot`, `AnchoredSpace` and `SumAggregator` can be seen. The SumAggregator is used to collect return values and sum them up. The main program can be seen in lines 18-36. At first the local `Robot` object joins the group (line 21). Then the `AS` is spread within 3 hops from node 5 (line 25). Once this spreading is complete, group operations can be carried out. This is done in line 29, where all robots are turned for 3 milliseconds. In this line, a method called `m2f`() is used. This method converts an arbitrary method call with its actual parameter(s) into a function object (so called functor). This functor is then propagated across the object space. Line 34 shows a synchronous call, where the calling robot is blocked until all participants have answered or a timeout occurs. The result is saved in the `SumAggregator`, from which it can be read at any time later. This is done in line 35, where a second value is also obtained from the aggregator. This value is the number of participants, which could be used to evaluate the validity of the first value. The number of participants could be recorded for every group operation, and if it falls below a certain threshold, the operation could be repeated. It could also be used as an indicator for quality of service.

There are also some local operations implemented which can be used with an object group. As you have seen in the example you have to call the `join`()-method to connect an object to a group and you can call `remove`() to disconnect. The method `reset`() resets the local group object, this means a local connected object is disconnected and all local group topology information is removed. That implicates that the local node and all nodes in the subtree do not receive group operations any more. Because `reset`() is only a local operation, it has to be called on all nodes in the group to reset the topology. After this, the `init`()-method of the space can be used to create a new group topology.

4. Experiments

This section describes the experiments we made. As the simulations delivered perfect results, they are not described here. Only the much more interesting results of the real experiments are discussed. All our experiments were conducted on modified RCX robots, which have been additionally equipped with a radio module of type ER400TRS which we use instead of the included infrared module (IR) to enable broadcast protocols.

In all our experiments every node wanted to be a group member. The first experiments on the RCX robots were made in a single hop environment. 9 nodes were gathered together, each within at most 30 centimeters of all others. These Experiments delivered the same results as the simulations. Once the TDMA MAC had found its slots and no more messages were buffered all messages were received correctly. This led to nearly perfect results, as only the messages that were lost by the MAC had negative influence on the results. Once the single hop environment worked, we moved on to a multi hop environment.

For the multi hop environment the 9 nodes were arranged in a square of 3 times 3. The distance between nodes was 10 meters, figure 4 shows their layout. An

7	8	4
9	5	6
1	2	3

Figure 4. Layout of nodes

experimental run consisted of three parts. First, a routing tree was built. Then 10 asynchronous group operations were carried out and the number of processed operations on every node was noted (not shown here). Finally 10 synchronous aggregations took place. The number of local processed aggregations was noted, as was the result of the aggregations. These experimental runs were started from each corner and with 3 different spaces, the `ConnectedSpace` was ignored as its difference to the `AS` can only be seen in larger networks. As expected, the results differed quit a lot, depending on the space in use. For this reason, the results are shown separately for each space.

4.1. The ATBFloodingSpace

Figure 5 shows the number of locally conducted aggregations. Node 6 did not participate at all and node 9

10	02	09
01	09	0
07	09	09

10	10	***10***
0	10	0
10	10	10

02	10	10
0	10	0
10	05	10

01	08	01
09	09	0
03	09	***10***

Figure 5. Number of local aggregations, ATBFloodingSpace. The starter nodes are shown in italic.

participated only in the run started from node 3 (shown on the lower right) which was the first one made. The return value of each node was its identity, the aggregation was done as summation of all values. This means that a perfect run would result in a return value of 45 (sum of 1-9). As Nodes 6 and 9 did not participate due to hardware problems, a value of 30 would be good. Figure 6 shows the results of the 10 aggregation started from node 4 as example, the values in the first line are the result obtained within the calculated time, the second line was obtained after the timeout was already long past. Node 4 was chosen, because all participating nodes received all 10 aggregation messages. The message losses in the other runs can be explained with link breakages due to changing signal strength and with problems of the MAC-layer.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
a	30	20	04	27	04	04	17	22	15	30
b	30	20	28	27	28	28	22	22	15	30

Figure 6. Results of the aggregation obtained by node 4: a) within the timeout, b) thereafter

The results of the aggregations shown in figure 6 can be interpreted as follows. In 4 out of 10 aggregations the upper value which was obtained within the time limit calculated is smaller than the one obtained later. This means that the timeout on the starting node 4 was to small, because results still arrived after the timeout. The values on the lower row deliver interesting insights, too. Please remember that a value of 30 as was reached in aggregation 1 and 10 is optimal. In aggregations 3, 5 and 6 node 2 failed to deliver its result, in aggregation 4 this was the case for node 3 or for nodes 1 and 2. These are the nodes that are farthest away from node 4 and were probably connected trough multiple hops. This shows that the problem with the timeouts exists on intermediate nodes, too. Once the timeout on a intermediate node arrives, the results gathered until that moment are sent back to the originator and the aggregation object is destroyed. Results that arrive later are ignored by the intermediate nodes and can not arrive at the originator of the aggregation.

4.2. The AnchoredSpace

The `AS` builds a routing tree first, which is then used in all following group operations. This building of the tree is crucial for the rest of the experimental run, as bad connections can have a strong negative influence on the overall performance. If a link is used during the construction that breaks soon after, the whole subtree can be lost. Figure 7 shows such a bad example. Node 8 is a physical neighbor of node 4 which started the run but it is connected via 3 hops. The reasons for this can be as follows: The reception of node 8 was bad when node 4 started building the tree so it missed the message from node 4. That it is not connected to node 5 instead can be for the same reason, or it can be the TDMA MAC's fault. If nodes 3 and 1 were able to send before node 5, node 8 received the message from node 1 first.

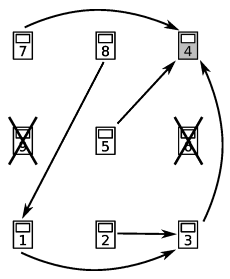

Figure 7. Topology of an AS initiated by node 4. The arrows point at parents

Nodes 3 and 7 are 20 meters from node 4 but still they are connected to it. This longer distance means that the links will break more often. In figure 8 one of the big problems of this multi hop environment can be seen. Node 3 received only 7 aggregations. Consequently, it was only able to forward these. While node 1 received all 7, node 8 received only 5. The number of received aggregations can only decrease on each hop, leading to bad responsiveness of the nodes that are connected over too many hops.

10	5	***10***
0	10	0
7	7	7

Figure 8. Local aggregations

The results of the aggregation were such as could be expected. In nearly all aggregations only node 5 was able to answer in time. Another problem that has not yet been addressed is that the parent node returns its value when it thinks that all children have answered. In this experiment, the message telling node 4 that node 3 is its child has been lost. Thus, node 4 only waits for 1 child which is node 5 before returning its value. This means that within the timeout no value higher than 9 could be reached. But even after the timeout node 3 was not able to answer except for cases 3 and 6(figure 9). In case 3 all values arrived after the timeout, in case 6 only node 8 is missing.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
a	9	4	9	9	9	9	9	9	9	9
b	9	9	30	9	9	22	9	9	9	9

Figure 9. Results of the 10 aggregations: a) within the timeout, b) thereafter

These results tells us two things, the first is that the timeouts are too small for multi hop. The Second is that the `AS` should only be used in networks with links that are stable for longer periods of time or in single hop environments. As these are the scenarios it was designed for, the experiments confirm its suspected inhibitions in other scenarios.

4.3. The RobustAnchoredSpace

The `RAS` addresses the problem of lossy links by using a simple retransmission protocol. Nodes that have sent their return value wait for the parent to acknowledge it. If no acknowledgement is received within a certain time, the value is sent again up to x times. X is configurable and was set to 3 in our experiments. For comparability reasons the results are shown for the run from node 4.

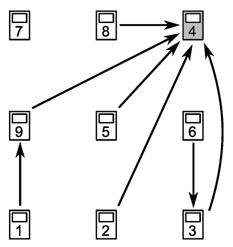

Figure 10. Topology of a RAS started from node 4. The arrows point at parents

Figure 10 shows the topology the `RAS` created. Node 7 was not able to connect, even though all messages were retransmitted 3 times. However, even nodes 6 and 9 participated this time. Node 9 did so in all our experiments with the `RAS`, node 6 in half of them. This shows a robustness gain already, as these did not participate in any of the experiments with the `AS`.

0	10	***10***
7	9	10
6	8	10

Figure 11. Local aggregations

The number of local aggregations can be seen on figure 11. This time, node 9 received more messages and thus was able to forward more. Nodes 2 and 5 worked slightly worse than before.

Figure 12 shows the results of the 10 aggregations. None of them reached the optimal value but nearly all of them are better than the results from the `AS`. Still, even with the longer timeouts due to the retransmission window, the results returned after the timeout has long passed are much better than those within the time limit. This shows again that the global timeouts are too short. Also, the timeout for retransmissions is another crucial factor, which has to be tuned. When messages are buffered in the MAC layer and all retransmissions occur before even the first message is sent, the retransmissions have gained us nothing.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
a	12	4	28	19	21	12	19	12	4	9
b	12	23	28	22	21	19	22	28	28	26

Figure 12. Results of the 10 aggregations: a) within the timeout, b) thereafter

4.4. Summary

All spaces discussed here have a common problem: The cross-layer issue of timeouts with the MAC layer. If a MAC was used that could guarantee delivery times, the timeout would work much better. The problem is located in equation 1. The timeout for 1 hop is the only part of the equations that depends on the MAC, and we chose to set it to 3 times the frame size of the MAC. As the experiments have shown, that was to small for this scenario, and needs to be fine tuned in the future. Apart from that each space has its own strengths and weaknesses. Therefore it is not possible to say which of the spaces is the best, rather this question depends on the application. The factors affecting the choice of space are link stability, whether operations occur sporadic, in bursts or often and the targeted platform. If there is not enough memory available for the `RAS` it can not be used, even if it would otherwise be the perfect choice. Also, the number of sent messages can be a factor, as can be the frequency of topology changes. The selection of an `CS` is independent from these factors, as it is only application specific.

	frequency of operations		
	seldom	in bursts	often
stable links	ATB	AS	AS
unstable links	ATB	ATB	RAS

Figure 13. Choice of space

Figure 13 shows an example of different choices for two of the factors described above. If the operations occur only seldom the `ATB` is recommended. For burst behavior in a stable environment the `AS` can be used while the ATB should be used for unstable links. If the operations occur often, the ATBs flooding would represent to high a network overhead. Instead, an `AS` or an `RAS` would be used. They would build their routing trees only once. For node mobility only the `ATB` works properly, there is no need to distinguish between operation frequency or link stability. If only asynchronous operations are used in any scenario there is no need to build a routing tree and the ATB should be used.

5. Related Work

Abstract regions [7, 8] offer a tuple-space like communication abstraction. An abstract region consists of neighboring sensor nodes, which are a certain number of hops or meters distant from an anchor node around which this region is centered. Like COCOS, abstract regions are designed as parallel programs. There are 3 big differences between abstract regions and COCOS. First, the membership of a node in one of COCOS's spaces can be determined by any factor, even the current sensor value. Thus, it is highly dynamic and can change at any time while the membership in an abstract region is static. Second, COCOS offers control over actuators or any other item on the sensor nodes additionally to the in-network aggregation which is provided by abstract regions. The last difference is the operating system used for the implementation. COCOS uses REFLEX as basis, which is an event driven operating system implemented in C++. This enables the usage of standard tools for both REFLEX and COCOS.

In wireless sensor networks, in-network aggregation is often realized by routing protocols, e.g. Directed Diffusion [4] and Rumor Routing [2]. There are two differences between these approaches and ours. First, their approach works on the network layer while our approach realizes a middleware that can be used in combination with any routing protocol. The second difference is that our approach is not restricted to the aggregation of values but rather enables the concurrent control of sensors and actuators on all nodes.

6. Conclusion & Future Work

The different spaces of COCOS enable distributed control over sensors and actuators. They also offer in-network processing of values by aggregating them with any given aggregation function. We have shown that the usage of remote method invocation mechanisms and sensor spaces is feasible and useful. When using the sensor spaces, application programmers need less time to program a sensor network, as they can program it as a whole rather than individual nodes. With the abstractions offered by COCOS it is easy to introduce QoS into in-network aggregation, as the number of participating nodes can be counted as seen in the example in chapter 3. Paradigms like publisher/subscriber can also be realized with minimal effort. One conclusion that can be taken from the experiments is that most simulations do not come close enough to the real world. In the simulations, a logical topology as seen in figure 7 where a physical adjacent node is connected over 3 hops never occurred. This convinces us that real experiments are absolutely necessary.

In the future more work on the timeouts is clearly needed as finding the correct timeouts is vital for the spaces to function properly. This is non trivial as cross-layer issues with the MAC and possibly with other layers as well have to be considered. There is also potential for more spaces. The AnchoredSpace and ConnectedSpace have shown two problems which are the building of the routing tree and the return of the values. The RobustAnchoredSpace and the ATB-FloodingSpace each address one of these problems, but a solution to both is still missing. If node density is high enough every node is reached by the ATB but the return values are sent exactly once to the parent. If the application needs a CS but the links are lossy, a retransmission and transport layer from COPRA can be used at the moment, but some kind of RobustConnectedSpace could be useful.

To make the initial routings trees for the AS, CS and RAS better, children could use a metric to determine their parent. At the moment, the parent is the node, whose message arrived first, if another message arrives later it is simply discarded. If some value for the link quality could be obtained, e.g. signal strength, a message received from a better link could override previous messages.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proc. of 24th International Conference on Distributed Computing Systems (ICDCS)*, march 2004.

[2] D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 22–31, New York, NY, USA, 2002. ACM Press.

[3] I. Chatzigiannakis, S. Nikoletseas, and P. G. Spirakis. Efficient and robust protocols for local detection and propagation in smart dust networks. *Mob. Netw. Appl.*, 10(1-2):133–149, 2005.

[4] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[5] R. Karnapke and J. Nolte. Copra - a communication processing architecture for wireless sensor networks. In *Euro-Par 2006 Parallel Processing*, pages 951–960. Springer, 2006.

[6] K. Walther and J. Nolte. Event-flow and synchronization in single threaded systems. In *Proceedings of First GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES)*, Mar 2006.

[7] M. Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[8] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42, 2004.