Adapting Paintable Architecture Concepts to Wireless Sensor Networks

Marin Orlić, Igor Čavrak, Mario Žagar
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
marin.orlic@fer.hr ; igor.cavrak@fer.hr ; mario.zagar@fer.hr

Abstract. *Amorphous computing promises a novel approach with massively distributed systems. Research in the field hasn't yet produced any formal methodology for design of such system.*

Paintable computing architecture provides an interesting testbed for process self-assembly and together with wireless sensor networks, presents a platform whose goal is to tightly integrate with its environment. Such a platform could combine the best of both technologies while running applications composed of fragments resembling classical software agents capable of mobility.

This article presents an introduction to changes that are necessary to have all these similar computing approaches unite.

Keywords. Amorphous computing, wireless sensor networks, neighbourhood data sharing, *paintable* computing.

1. Introduction

The fact that, when properly connected, large numbers of processing nodes – although simple for themselves – produce results comparable to much larger machines, permeates most of the scientific and industrial projects in the last few decades. Networking itself holds no inherent limitation to the size of nodes – if there is no need for nodes to interact with humans directly, nodes of any practical size can be imagined – from huge communications centres to breadcrumb sized micro-units dispersed in any conceivable environment.

The idea of amorphous computing emerged from the possibility that some day a networking node itself could be just a small speck capable of autonomous functioning, and for thousands of such nodes to perform complex tasks.

Amorphous computing is defined as the effort to produce engineering principles and languages that can be used to observe, control, organize, and exploit the behaviour of programmable multitudes. The objective of this research is to create the system-architectural, algorithmic, and technological foundations for exploiting programmable materials [1]. Such materials can be considered biological in the same sense that biological organisms are formed from multitudes of relatively simple cells running genetic programs shared by the whole colony.

Our goal is to produce such a system by combining two of such paradigms – *paintable* computing with its biological inclination and wireless sensor networks with their capability to react to environmental events. Some ideas for adaptation of *paintable* concepts to sensor networks will be presented, with the goal of making it simpler to run (mobile) agents on this platform. We believe it is possible to adapt *paintable* methodology to be used on slightly different hardware of wireless sensor networks (WSN). Hopefully this effort would help to simplify programming and use of WSN in complex applications.

This article will explain basic properties of the *paintable* platform with respect to the possibility of their implementation and adaptation to WSN platform. *Paintable* architecture cannot be directly implemented on WSN; however some of the concepts presented can be used to reduce programming complexity. Neighbourhood abstraction used in *paintable* could is a valuable framework for WSN applications – every application running on distributed sensors needs a method for neighbour discovery and local data sharing. *Paintable* design shows that it is possible to use localized data to construct application scaffolding throughout the entire particle ensemble.

Code mobility provides a different perspective to application development, similar to agency. Agents could perform sensor data-aggregation and calculations locally. Instead of using generic aggregation schemes, every application creates specialized agents for this task increasing overall efficiency.

Student Paper

2. Paintable architectures

Basic principles of a *paintable* computer have been presented in [5], following the idea that advances in process technology will soon enable production of autonomous computing elements the size of sand grains. Such computing elements, call them *particles*, could easily be suspended in any medium, provided they are resilient to environmental stress – even *painted* on some surface (as the name of the architecture suggests), rendering it to some extent intelligent.

Basic architecture of a *paintable* particle described by [5] consists of a microprocessor with some memory and wireless transceiver (or some other kind of networking device) contained in a single package and powered parasitically. Once powered, particles boot and organize themselves, forming complex structures and performing complex tasks. Such a system resembles biological systems, using multitudes of unreliable elements in unknown arrangements growing into precise forms and behaviours.

Table 1 lists typical characteristics of a single particle. Systems with such specification can easily be constructed even today, with main restraints being the communication (network) and power subsystems. Communication subsystem's effect on power usage is drastic – unlike in larger systems, most of the power on small devices like WSN is used for communications, with CPU power usage an order of magnitude smaller.

Theoretical paintable implementation has no issues with power – ideally it should power itself by harnessing environmental power, giving it unlimited life span with respect to power. In contrast, existing battery-powered devices life span is in range of months.

Table 1. Typical characteristics of a *paintable* particle

Subsystem	Specification	Comment
Processor	'486-class at ~50MHz	
Memory	50-100K words RAM + OS in ROM	Code/data storage
I/O	Wireless, duplex at >100Kbps	Broadcast
Power	Environmental	Harvests power from immediate environment

Essential properties of *paintable* making it interesting for adaptation are [6]:

- self-assembly of unreliable (and cheap) computing elements – particles, solving issues plaguing complex systems like:
  - fault tolerance,
  - adaptive topology,
  - complex system combinatorics,
  - clock asynchrony between particles.
- relatively simple particle OS (or toolkit),
- program model based on relatively simple process fragments (called *pfrags*) capable of mobility between particles

Pfrags are described as autonomous, mobile program entities capable of sensing and reacting to their environment [5], a definition that closely resembles that of a software agent.

Particle memory is segmented into areas – *pfrags* run entirely from RAM memory and contain all necessary buffer memory inside their payload (*pfrags* are mobile entities). I/O space of the particle is memory-mapped into pages, with local HomePage acting as a proxy for communication with neighbouring particles. All data entered into local HomePage is mirrored to all nearby particles where it appears in their I/O space. This mirroring process is done by the networking subsystem. *Pfrags* are allowed to communicate data to other particles only through posts in their local HomePage, thus simplifying the system complexity from developers' standpoint.

Particles have no ROM for static *pfrag* storage – all *pfrags* in the system must enter through I/O portals, external devices masquerading as particles. *Paintable* is therefore completely dynamic, the only static code is the OS and any embedded OS toolkit functions needed for basic particle functioning.

Paintable is designed to handle inherent unreliability of particles themselves, implying very large number of particles. The dynamicity of the system makes it dependant on I/O portals loading the application fragments into the ensemble. Network usage during application deployment is therefore very high until the ensemble reaches the stable phase and *pfrags* stop migrating between particles. Even with very limited networking abilities, it is conceivable that particle's neighbourhood size can be a problem. HomePage mirroring requires a lot of memory, effectively reducing either the amount of data that can be shared, or the maximum neighbourhood size. The adaptation capacity and fault tolerance can increase resource usage and

greatly reduce the possibility for its usage for complex applications.

3. Wireless sensor networks

Wireless sensor networks (WSN) are a propulsive field within pervasive computing, following the idea that distributed sensing environments can increase sensing reliability and responsiveness using local computing power 0.

WSN follows the same basic principle as *paintable* – sensors (motes) are (randomly) distributed nodes capable of only simple computing tasks, but unlike particles, motes are equipped with one or many sensors. This orientation to sensing guided basic construction principles for motes. outlines typical characteristics for a WSN mote.

One of key differences between mote and particle architectures is that motes provide plenty of ROM space, to be used for static program storage, and scarce RAM resources.
Unlike *pfrags* which are completely dynamic, motes allow only for static programs stored in ROM (some motes allow this code to be changed over the network).

Typical mote operating system is TinyOS, although some motes run on Linux. TinyOS is statically linked to program code and uploaded to motes' ROM memory. Program execution is split-phase – TinyOS supports sequentially executed code fragments called tasks, used for long running computations. All tasks are statically defined at compile time and posted for execution when needed. Active tasks are scheduled for execution on a FIFO basis. Running task cannot be pre-empted by another task, only interrupted by events (e.g. receipt of data from the network or timers). To simulate blocking I/O behaviour, an application should use state machines to schedule appropriate tasks for execution. Scheduling of multiple jobs (*threads*) is also application-managed.

Table 2. Typical characteristics of a WSN mote

Subsystem	Minimal	Typical (Intel iMote 2003)
Processor	Atmel at 4MHz	ARM 7TDMI at 12-48MHz
Memory	8K ROM, 0.5K SRAM	512K ROM, 64K SRAM
I/O	RF, 10Kbps	USB, GPIO, I2C, Bluetooth
Power	battery	battery

4. Adaptation of *paintable* concepts to wireless sensor networks

Paintable concept is an interesting experiment in amorphous architectures, but inappropriate for implementation regarding currently available hardware platforms.

Key software concept embodied in *paintable* architecture is that of dynamic, mobile programs (process fragments) streamed into the particle assembly through I/O portals. This is very difficult to implement directly on WSN platform. Motes do not allow dynamic code or partial changes to ROM contents. Avoiding this limitation could allow implementation of simple mobile agents running on WSN. Since I/O portals are impractical for systems running autonomously without human supervision, all necessary software would have to be preloaded on (at least some of) the motes and executed when the need arises. Identification of necessary *pfrags* to be preloaded on motes is therefore important, even more so if code mobility proves to be too expensive.

HomePage paradigm presented in *paintable* requires mirroring of entire HomePage contents from one particle to another which greatly simplifies the platform conceptually. Actual implementation of such an idea requires, however, several modifications – for instance, neighbourhood size should probably be limited in compile time due to the nature of TinyOS. System designers must make a choice between large number of neighbours or large amount of data in HomePage. Mirroring and synchronization of entire HomePage structures to and from neighbouring particles presents a large networking overhead, and severely damages performance in the long run, energy-wise. This is not a problem for *paintable*, with particles that are powered from the environment, but a major issue with WSN platform. Since most applications need to share only a subset of data between neighbouring particles, a modified version of HomePage with additional functionality allowing pfrags to mark data for sharing provides an increase in functionality.

Additionally, in case that the whole particle/mote ensemble runs multiple applications, or different modules of the application, with ensemble nodes dedicated to providing different functionality, the volume of mirrored data can be further reduced. Given that all neighbourhood communication is broadcast, network usage is already nearly minimal. If

HomePage module is further extended to allow applications to pick only relevant data from the entire broadcast dataset, cluttering memory with unnecessary data is avoided.

Basic modifications necessary for successful adaptation can be identified as follows:

- adapt software environment available on motes to allow dynamicity in code execution and some kind of code mobility,
- identify necessary *pfrags* and adapt them to WSN platform.
- adapt networking subsystem of WSN platform to provide neighbourhood integration similar to HomePage concept, but with integrated data sharing and filtering capabilities to increase memory usage efficiency.

Mentioned modifications will be discussed in detail in the following chapters.

5. Software environment modifications

Process fragments (agents) running on motes need to have some method of communication, both locally (within a single mote) and with other fragments situated on nearby motes. In the case of communicating small amount of data (e.g. to announce agent presence), the metaphor of HomePage used in *paintable* is very appropriate. To implement HomePage behaviour on WSN motes, broadcast network capability is required, and supported by TinyOS. Such broadcasts need to be localized – the size of the neighbourhood is limited by available local memory, since HomePage data from all neighbouring motes needs to be mirrored locally. Such behaviour is not desirable if larger amounts of data need to be transferred between agents, thus requiring additional messaging systems other than HomePage mirroring. HomePage posts can still be used to provide framework for agent synchronization. To provide uniform messaging, HomePage post creation can simply be extended to allow agents to prohibit mirroring for that post, declaring the post private instead of shared.

Code mobility and dynamic execution of agents could be performed in two ways, depending on the type of system environment. In case the system environment is static (in-the-field scenario), no new agents can be introduced into the system – I/O portals of some sort are used solely to gather data collected by the system and monitor and manage its performance. An open, dynamic environment would typically be used in 'friendlier' environments.

5.1. Static environment scenario

Static environment scenario could use motes with preloaded native-code agents of all types, and activate them when needed. Activation messages would then be the equivalent of mobile agent transfer and contain the agent state (in case of strong migration) or just activation requests (in case of weak migration), but no code, to reduce communications overhead. After activation message is received, platform would prepare the agent for execution, load its instance data and execute it. Forcing all motes to have every agent in the system preloaded in ROM poses a possible overhead in case an agent will eventually be executed on just a fraction of the whole mote ensemble. This can be reduced if motes with different combinations of preloaded agents are intermixed in sufficiently dense deployments to allow redundancy of motes with particular agents preloaded and mobility between these motes.

5.2. Dynamic environment scenario

Dynamic environment supports dynamic introduction of agents into the system, requiring some sort of virtual machine. A sophisticated virtual machine could perform its own scheduling, blocking operations, manage access to shared memory (HomePage) and data exchange between agents. Virtual machine approach leverages code mobility (smaller programs) with higher CPU usage while interpreting pseudo-instructions and slower execution. Smaller programs allow for energy savings when the system needs to be reprogrammed (an agent migrates to a new mote). Maté virtual machine (or virtual machine toolkit) [2] supports code mobility and is proven to be energy efficient when used with code that is invocated infrequently. Maté follows basic TinyOS structure, having execution contexts that run code capsules. Simplifying development, split-phase operations (command/event pairs in TinyOS) are replaced with blocking operations (just ***send*** instead of ***send***/***sendDone*** command/event pair). Different VM implementations based on Maté exists, and some support multi-agent scenarios (Agilla [3] supports up to 4 agents on a single mote). VM code is compact and easily transferred over the

network, but motes still need to periodically exchange code version information over the network to synchronize the distributed code-base. Complex applications pose a significant energy overhead when implemented entirely using any kind of virtual machine.

6. Framework design

Sample agent support framework in closed environment is outlined in Fig. 1.

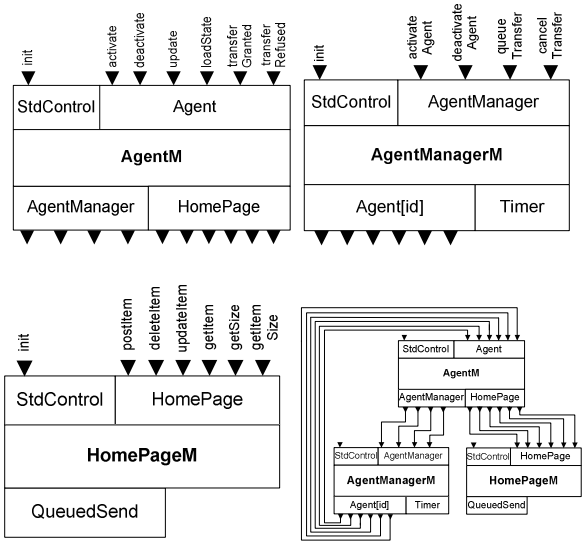

Figure 1. Agent support framework

Modules like AgentManagerM, AgentM and HomePageM are defined. AgentManagerM module provides services required for agent activation and migration (agent code is preloaded). AgentManager posts agent's update commands for execution on each timer event. AgentM functions as an interface for communication with each agent. HomePageM module provides toolkit functions (just a subset is shown) to support posting items on HomePage, and allows agents to register post names to be mirrored from the neighbouring nodes. When an item is created or updated, HomePage posts a task to synchronize that item's data with neighbours if item is marked as public and signals an event. Upon data receipt, neighbouring particles pick interesting posts (depending on current agent configuration of that node) and mirror them. This approach reduces the memory needed for HomePage storage, and increases the network efficiency, since only shared posts are broadcast. Neighbourhood (mirrored HomePage array) size is limited in compile time.

7. Selection of necessary process fragments

The introduction to paintable architecture in [5] describes several elementary pfrag components useful for construction of complex systems over *paintable*. Table 3 summarizes some of them.

Not all of these sample components will be necessary for crafting a particular system, but their presence provides a basic framework for implementing a wide range of complex applications. In addition to listed components, a family of messaging components will also need to be defined, to allow message-based communication between agents.

Table 3. Examples of self assembly on *paintable* [6]

Component	Uses *pfrags*	Description
Gradient	1	Estimates the shortest distance back to a fixed anchor point.
Diffusion	1	Accepts a data packet as a payload, seeks to position itself to minimize disparity in local density.
MultiGrad	1	Virtual pfrag emulates multiple Gradients.
Channel	3	Defines a bidirectional communication channel between two anchor points
Coordinate	5	Constructs 2D coordinate system from 2 anchor points.

Motes are easily addressable, however the location of particular mobile agent need not be constant after the message is sent. To facilitate tracking, each agent could emanate a gradient field (using MultiGrad component present on all motes). In case of a larger number of agents this would however require too much memory to track agents' field strengths and overload the motes.

8. Related work

Paintable architectures haven't been used solely in simulated experiments of self-assembly. Although forming of complex structures requires large number of particles, some interesting applications can be tested even with relatively small particle count. Concepts like localization (particle ability to determine its own position

within the assembly) and clock synchronization over the assembly were demonstrated with pushpin platform [6]. A sample implementation of *paintable* architecture, pushpin particles receive power and connectivity through layers of conductive materials in testbed's surface.

Pushpin ensemble implements a modified version of paintable platform, addressing some of the issues outlined in this article.

Complexity and lack of formal methods for analysis of amorphous systems increase the need for actual hardware, real-world implementations. Tested in real implementations, knowledge of applied heuristic methods will eventually assist to form formal methods, enabling engineering of complex adaptive amorphous systems, just like any other technical system. We hope that the effort to adapt the architecture to easily available hardware is the step in the right direction.

9. Future work

Future work will be directed to solving problems outlined above to assist with an actual implementation of mobile agents on wireless sensor platform using neighbourhood abstraction through HomePage. Issues of concern will be:

- implementation of HomePage-like concept on mote architectures (blackboard-like neighbourhood shared memory),
- implementation of agent hosting framework, as described in static environment scenario section,
- adaptation of basic self-assembly *pfrag* toolkit (similar to those sample *pfrags* noted above) to TinyOS platform.

With extensive use of neighbourhood for reflective data storage and inter-process communication, pushpin is an interesting testbed. Practical implementations of its concepts are necessarily limited in resources, and some modifications are required. WSN platform is designed with many constraints in mind, some of them severely complicating application development. A framework providing facilities similar to those on *paintable* would allow theoretical benefits of *paintable* to be tested on real-world platform and compared with existing applications

10. References

[1] Abelson H, Allen D, Coore D, Hanson C, Rauch E, Sussman GJ, Weiss R, Homsy G, Knight TF, Nagpal, R. Amorphous Computing. Communications of the ACM, vol. 43, 2000: p. 74-82.

[2] Levis P, Culler D. Maté: A Tiny Virtual Machine for Sensor Networks. Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002; San Jose, California.

[3] Fok C-L, Gruia-Catalin R, Lu C. Mobile Agent Middleware for Sensor Networks: An Application Case Study. Proceedings of 4th International Conference on Information Processing in Sensor Networks; April 2005, Los Angeles, California.

[4] Kahn JM, Katz RH, Pister KSJ. Next Century Challenges: Mobile Networking for "Smart Dust". Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking; 1999; Seattle, Washington, United States; 1999. p. 271 – 278.

[5] Butera W. Programming a Paintable Computer. PhD thesis: Massachusetts Institute of Technology; February 2002.

[6] Paradiso, J, Lifton J, Broxton M. Pushpin Computing. Responsive Environments Group, MIT Media Lab. Massachusetts Institute of Technology. http://web.media.mit.edu/~lifton/Pushpin [02/20/05]

[7] Lifton J, Seetharam D, Broxton M, Paradiso J. Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks. In: Mattern F, Naghshineh M, editors. Pervasive Computing, First International Conference, Pervasive 2002, Proceedings; 2002 August 26-28, 2002; Zürich, Switzerland; 2002. p. 139-151.

[8] Pottie GJ, Kaiser WJ. Wireless Integrated Network Sensors. Communications of the ACM, vol. 43, 2000: p. 51-58.