

An Architecture and Protocol for the Management of Resources in Ubiquitous and Heterogeneous Systems based on the SVP model of Concurrency

Chris Jesshope¹, Jean-Marc Philippe², and Michiel van Tol¹

¹ University of Amsterdam, Institute for Informatics
Kruislaan 403, Amsterdam 1098 SJ, Netherlands

² CEA LIST, DRT/DTSI/SARC/LCE, CEA-Saclay Batiment 528
Point Courrier 94, F-91191 Gif sur Yvette Cedex, France
{jesshope,mvantol}@science.uva.nl, jean-marc.philippe@cea.fr

Abstract. This paper proposes a novel hierarchical architecture and resource-management protocol for the delegation of work within a ubiquitous and heterogeneous environment. The protocol is based on serving SVP places to delegate a component of work together with the responsibility for meeting any non-functional computational requirements such as deadline or throughput constraints. The protocol is based on a market where SANE processors bid for jobs to execute and selection is based on a cost model that reflects the energy required to meet the jobs requirements.

Key words: concurrency models, heterogeneous systems, resource management, market models, ubiquitous systems.

1 Introduction

As CMOS nodes continue to shrink, the complexity of embedded systems grows. This progress enables the manufacturing of low-power and low-cost consumer electronic devices able to communicate through wired or wireless technologies. Embedding computing power in everyday consumer product leads to the possibility of having systems comprising networks of thousands of nodes near each user. This will provide everyone with the possibility of processing data any where at any time, moving people into the pervasive computing era [1].

The design of such systems requires a dramatic shift at every level of the system as neither software nor hardware platforms are ready to face the issues raised by this exciting new research challenge. These ubiquitous systems may comprise a huge number of heterogeneous computing elements and will evolve around the users following their needs and habits. Thus, their optimisation will be highly dependant on their computing environment. Taking advantage of the huge computing power offered by this collaboration of elements will require the

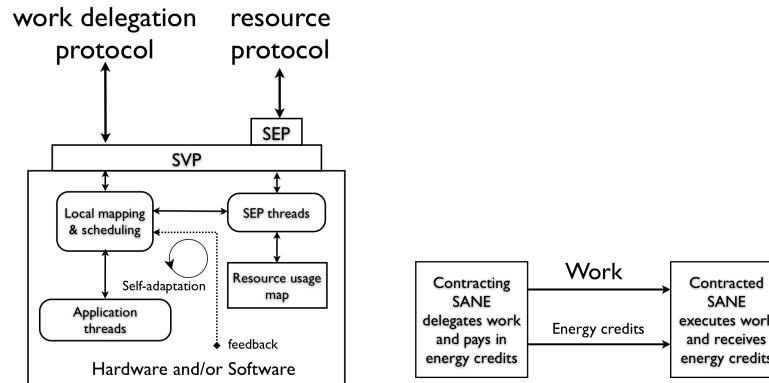


Fig. 1. Generic SANE (may be a collection of SANES) responds to two protocols: one to perform work as families of threads the other to serve resources to external threads. The latter uses negotiation between SANES based on energy credits

dynamic management of concurrency, including graceful degradation under conditions where computing elements may appear and disappear at will. This is a significant challenge.

To solve these issues, a disruptive approach is being promoted in the *ÆTHER* European project, which embeds self-adaptivity at each level of the system³, giving autonomy to the components and enabling the application designer to concentrate on the application instead of having to cope with all possible events in the lifetime of a computing resource in such a rapidly evolving environment. For this purpose, we have introduced the SANE concept (Self-Adaptive Networked Entity). This views the system as a collection of self-adaptive elements (software, hardware or both) that can observe their environment and their internal performance so as to autonomously modify their behaviour in order to improve the overall system performance. These elements collaborate with each other and share information and resources in order to provide a global optimisation based on local and autonomous behaviour. This approach requires a new architecture and protocols to enable the dynamic sharing of resources and the consequent management of concurrency.

The mechanism that enables this distributed sharing of resources is the delegation of responsibility for the execution of units of work, where that responsibility includes meeting performance constraints. We consider here, a hierarchical cluster-based architecture, where each cluster presents a uniform interface to its environment defining it as a SANE processor (or cluster of SANES) to be a SANE it must support the SVP model (SANE Virtual Processor) [2], see Figure 1. This paper describes the resource management protocol that enables delegation of work. SANES are autonomous and from time to time may be given jobs to execute; a local user may submit a job or one may be delegated from its

³ More details can be found on the projects web site: <http://www.aether-ist.org/>

environment. In the latter case, the SANE will have contracted with an external thread to run that job and to meet certain expectations in its execution, for example performance. The contract is negotiated using a credit exchange, where the cost of executing a job is initially assumed to be the energy expended by the contracted SANE. This can be measured in Joules. The contracting thread, which may be acting on behalf of another SANE, transfers credit for the agreed amount of energy to execute the work on the contracted SANE. In response, the contracted SANE agrees to meet the deadlines or performance constraints imposed by the contracting SANE.

2 The SVP model and its resources

SVP is a concurrency model that defines a number of actions to enable the execution and control of families of identical blocking threads. It is a hierarchical model and any SVP thread may create subordinate families of threads. The family (and its subordinate families) is the unit of work that is delegated in a SANE system. Implementations of the SVP model have been demonstrated and evaluated in software [3], based on the *pthread* library and in hardware [4], based on instructions added to the ISA of a many-core processor. The SVP model is captured by the five actions listed below and their implementation will define the underlying protocol supporting the interfaces defined in Figure 1.

1. *create* - creates a family of indexed threads at a place with parameters $\{start, step, limit\}$ defining the index sequence. It is based on one thread definition and returns a family identifier that uniquely identifies that family for asynchronous control of its execution.
2. *sync* - blocks until the specified family of threads and all of their writes to memory have completed. It returns an exit code that identifies how the family terminated; in the case of break, it also returns a value from the breaking thread and in the case of squeeze, it returns a family index value.
3. *break* - only one thread in a family can succeed in executing a break, which terminates its family and all subordinate families. It returns a break value of a type specified by the thread definition to the family's sync action.
4. *kill* - asynchronously terminates a specified family of threads and all its subordinate families.
5. *squeeze* - asynchronously terminates a specified family of threads and any user-specified subordinate families so that it can be restarted at the squeeze point, which is returned via each squeezed family's sync action.

SVP has two essential roles. At the hardware level, it captures locality and regularity, which are key factors in mapping a computation to a set of resources, whatever they are. The mapping defines wires or synchronisers to support blocking and these must respond on a hardware timescale. Constraints in the SVP model capture this locality, which reflects the asynchrony and locality that will be required in future silicon systems. The model expresses this by constraining communication between blocking threads. The first child thread created may

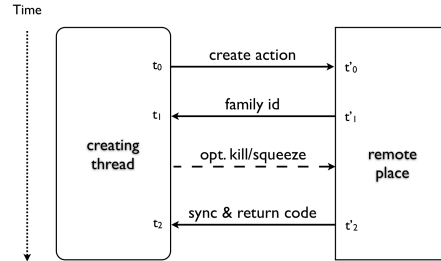


Fig. 2. Illustrates the SVP protocol between a thread and a place

synchronise only with the parent thread and other created threads their predecessor thread in the family. The model, rather than the program, exposes this to the compiler in order for it to statically map a computation onto hardware, using knowledge of the target implementation. Examples are compiling the language μ TC to a multi-core ISA [5] or mapping and routing a family of SVP threads to FPGA hardware. Using novel self-adaptation techniques, these SVP hardware threads may be dynamically optimised using online-routing [6]. In both cases, the implementation will be captured as one or more binary modules that support local communication.

SVP's second role captures the dynamic distribution of work between different implementations of the SVP model. This is achieved by binding an abstract resource to a unit of work on the creation of a family of threads. That resource abstraction is the SVP *place* which is provided by a place server. An implementation of place provides a network address and a token for authentication when creating work there. For example, when a place is served, the address is used to implement the protocol, in whatever network setting the SANE exists. More importantly, to avoid unauthorised use of a place, the place server gives both the place and the thread requesting it a token, which must be matched during the SVP create protocol. Figure 2 illustrates the events in this protocol. It should be noted that the create action in this role is a form of a remote procedure call.

The use of place as an abstraction allows the dynamic binding of resources to code when creating a family of threads. The place also identifies a contract between two SANEs when delegating work, as illustrated in Figure 1, and hence it identifies a set of resources or virtual resources on which the work will be executed. This may be a partition of a multi-core chip, it may be a domain in an FPGA chip that is dynamically configured to execute the family of threads or it may even be a processor or cluster of processors in a Grid. Each will have its own implementation of the SVP actions and tools to compile μ TC into that implementation. To achieve this abstraction, every implementation of SVP must deal with two pre-defined places and variables of type place:

- The *local* place is used to tell the SVP implementation that all threads in this family should be kept local to the creating thread, which may have different interpretations in different implementations.
- The *default* place is resource naive and will be determined by mapping and scheduling algorithms of the SANE implementation.
- A place variable has a meaning dependent on the specific implementation of SVP. It is set by a place server and used as a parameter of the create action.

The place concept is a heavily overloaded: it identifies a contract between a thread and a SANE, which will specify a level of service; it also embeds an address and a security key, which are used in the implementation of the create action to delegate the work. Once a SANE receives some delegated work, locally that work becomes resource naive and will be mapped and scheduled by the local mapping and scheduling threads (see Figure 1). These threads use the place to identify the contract negotiated and hence locate the specific constraints on execution agreed to. They must then organise the work to meet the constraints on the contract.

3 Resource negotiation in SVP

The aim of SVP is to give a concurrency model that is as ubiquitous in its application as the sequential model. The two roles of SVP described above reflect a separation of concerns between algorithm design and concurrency engineering. Resource-naive SVP code is similar to the sequential model in that it has properties of determinism and deadlock freedom under composition. An SVP implementation is therefore free to map and schedule threads as it likes. However, when introducing resources via places, i.e. introducing concurrency engineering, it is necessary to introduce non-deterministic choice, to support broadcast and to provide graceful degradation. All of these issues are discussed below before the resource server protocol is presented.

Mutual exclusion in SVP. Non-deterministic choice is required to manage exclusivity of resource use in a distributed environment. The place server must offer its service to a number of client threads that all compete for the available resources. This is achieved in SVP by providing mutual exclusion at a place rather than in memory, which is asynchronous. A mutually exclusive place sequentialises concurrent requests to create a family of threads. As places abstract resources, this is just another overloading of the concept of place that can be mapped to its implementation. For example, in the *pthread* implementation of SVP [3], a mutually exclusive place simply uses a mutex. In the ISA version of SVP [4], mutual exclusion in a single processor is implemented by class bits in the place variable and corresponding state in the processor. The state indicates whether an exclusive family of that class is currently executing and hence sequentialises create actions in any of the classes. The resource management protocol is called the SEP interface and is a mutually exclusive place (the *System Environment Place*) at which external threads create the protocol's threads to request and obtain places for their exclusive use in the delegation of work.

Broadcast in SVP. Because SVP is a deterministic model, which does not include any communication primitives, broadcast in the model must be implemented as a create action to one of a number known of places. For example, if a SANE cluster comprised n SANEs, where each SANE provided an SEP interface at a place, which was stored in the array of places `SEP_cluster[n]`, then the μTC code below would broadcast a request to each SEP interface in the cluster. N.b. the create parameters are: (family id; place; start; limit; step; block) followed by a thread definition. In this code, n threads in family fo are created locally, each of which creates an `SEP_request` at an SEP interface.

```
int n; place SEP_cluster[n]; family fo;
...
create(fo;local;0;n-1;1;){
    family fi; index i;
    create(fi;SEP_cluster[i];0;0;1;) SEP_request(...);
    sync(fi);
}
sync(fo);
```

Graceful degradation in SVP. Now consider what happens to this code if one of the SANEs in the cluster suddenly drops out before completing the request. The code deadlocks, as one thread in family fo will wait forever for its sync and hence family fo will never complete. One solution to this, and in general for any situation that requires graceful degradation, is a time-out on the create action, which allows family fo to wait a finite time before it completes. This can be implemented using a time-out thread, which kills family fo after a given time.

4 Resource management protocol

The implementation of the resource negotiation protocol in a SANE environment, like the SVP protocol over which it is implemented, is dependent on a SANE's level in its hierarchy. The generic protocol must provide for the requirements of systems at many different levels, from chip to board level and at many levels in a network hierarchy. The protocol comprises five stages: announce, request, bid, agree, delegate. Specific implementations may omit stages that are implicit in the design at that level. For example, the first stage requires a SANE processor to announce its capabilities to the rest of the system. In an on-chip environment, the capability of each SANE processor may be known a-priori and this stage may be omitted. However, for a SANE processor at the board level attached to a network or coming into range of a wireless network, this stage would be mandatory.

Announce. In the first stage of the protocol, a SANE joining a cluster announces its capabilities using a common format for defining both resource capabilities and requirements. The protocol uses the concept of a *root SEP*, which is not necessarily a single, fixed place but an place variable via which all resource negotiation takes place. The root SEP and its possible implementations

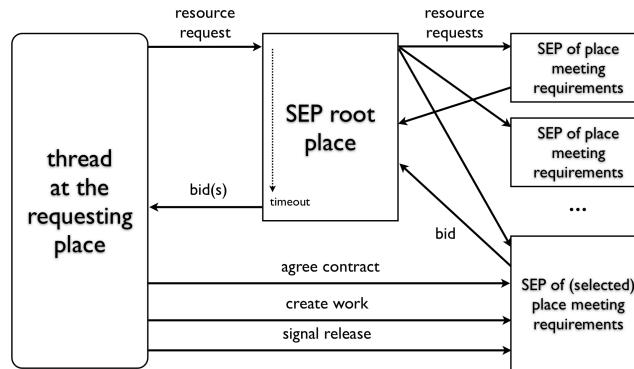


Fig. 3. Remainder of protocol, i.e. request, bid, agree and delegate, is undertaken when a thread requires resources to undertake a computation

are described in more detail in Section 5. On joining a network, some low-level communication protocol will first be established and on top of that a protocol for implementing SVP. The latter will initialise the joining SANE with a place to initiate the SEP protocol; that place is the root SEP and is similar in concept to the router in a conventional network. The joining SANE announces its arrival by creating the SEP-announce thread at the root SEP. Only one parameter is required, which is a pointer to the record(s) defining its capabilities. Those capabilities are defined using a domained ID that defines a set of known functions on the network. The domained ID serves to identify the processing domain of the work (signal processing, image processing, etc.) and the particular function offered or required. The root SEP can filter any requests for resources by the capability requested and hence reduce the amount of communication required. It does not make sense, for example, to send a request based on image processing to a SANE that does not implement any image-processing algorithms. The capability is defined as a processing rate on this set of functions. Note that the domains may represent functions at various levels of granularity, i.e. from arithmetic operations to complex functions. This step is illustrated in the μ TC code below. The SANE may also withdraw its capabilities from the pool using the `SEP_withdraw` thread. Of course it may also be withdrawn in a less graceful manner!

```
place root_SEP; family f_ann;
struct capability* my_capability;
...
create(f_ann;root_SEP;0;0;1;;) SEP_announce(my_capabilities);
```

Request. Having announced itself to its environment, a SANE may now make or receive requests for resources. These requests are again made to the root SEP, which will in turn forward the requests to any SANE in the environment that

is capable of meeting the request. This is defined as a required performance on a given function but also includes an elapsed time for which the resources are required. A timeout is attached to each request, which is the validity of the invitation to tender from the contractor. The request (and subsequent bids) are identified by the family identifier of the thread created in making the request.

Bid. Each bid will provide the following: a yes/no answer to the request and if yes, it will provide the overall cost for meeting the request, the time required to configure the resources, a lifetime (the provider will reserve these resources for this amount of time), the SEP to which agreement must be sent and a limit on the time the provider is able to provide resources, which may be less than or greater than the elapsed time requested. The cost can be in any agreed units but the use of energy allows the optimisation of the complete SANE system based on a (time, energy) couple. This step is illustrated below in μ TC and illustrated in Figure 3.

```
place root_SEP; family req f;
struct resource* my_request;
struct bid* my_bid, *good_bid;
...
create(req;root_SEP;0;0;1;;) SEP_request(my_request, my_bid);
```

Accept. When the thread requesting resources receives the list of bids, it will select one or more bids to meet its requirements and agree a contract with the provider. In response, the provider will return a place that defines the contracted resources. The family identifier of the initial request for resources identifies the contract. This stage is equivalent to signing the contract and, in a full market system, will result in a credit transfer from the requesting SANE to the providing SANE.

```
place root_SEP work_place; family f_req, f;
struct bid* my_bid, good_bid;
...
create(f;good_bid*.place;0;0;1;;) SEP_agree(f_req,work_place);
```

Delegate. All that is left to do when the `work_place` has been returned is to create the delegated work at that place and to signal the release of that place when that work is complete.

```
place root_SEP work_place; ; family f_req, f;
struct bid* my_bid, *good_bid;
...
create(f; work_place;;;;) my_work();
...
sync(f)
...
create(f;*good_bid.place;0;0;1;;) SEP_release(work_place);
```

5 The root SEP

The root SEP is a conceptual place and admits many different implementations. It is first and foremost, the place to which a SANE announces itself and to which it directs requests for resources. It is assumed that directly or indirectly, all known SANEs in a cluster may be reached from this place. Two examples of its implementation are given below that illustrate the range of possibilities.

A unique root SANE. The root SANE is the physical root of the cluster and is given responsible for maintaining a complete picture of the capabilities of all SANEs that have announced themselves within the cluster. It also provides an interface to the next level of hierarchy, which is called the environment in this paper. In this case, the implementation is trivial, at initialisation this SANE provides any joining SANE with its `root_SEP` which is then used as a target for all announce and request threads. The only problem in this implementation is that it relies on the root SANE being fault tolerant, as it is a single point of failure in the entire system. Note that if a single root SEP becomes overloaded, its resources can easily be partitioned and allocated to two root SANEs know by two subsets of SANEs.

Every SANE is the root SEP. Here, every SANE in the cluster receives announcements from all SANEs joining the cluster. In this case, on initialisation, each SANE must receive the SEP of all SANEs in the cluster and is responsible for announcing itself to all of them. Now it can broadcast its own requests to the cluster. This solution has maximum redundancy.

Other solutions provide various forms of partitioning, e.g peer-to-peer style approaches, where a particular SANE may know only of its immediate neighbours and where broadcast may proceed in multiple hops over subsets of the cluster.

6 Related work

The use of a distributed protocol for problem solving is not new. In 1980, Smith proposed the contract net protocol to specify distributed communication and control in a loosely-coupled, problem-solving environment [7]. In this protocol, task distribution uses a negotiation process, where an exchange of messages between nodes in the system decides which tasks are to be executed and which nodes may be able to execute those tasks. This protocol (and other work within *ÆTHER* [8]) adopts a managed approach to work delegation, i.e. one node, the manager, assumes responsibility for monitoring the execution of a task and processing the results of its execution. In the approach described here, both execution and the responsibility for meeting any execution constraints is delegated.

Although we adopt a market model here, the focus of this paper is the architecture and protocol. Also, the market is only required to provide deviation on a cost based on energy, where the market provides a distributed mechanism to detect and react to load. Further information on market-based resource allocation can be found in the following thesis [14].

Mapping and scheduling workflows (a set of tasks with sometimes complex dependencies) onto grids, e.g. GridFlow [9] and Nimrod-G [10] has similar requirements. Here, a more pragmatic and coarse-grained approach is adopted, based on job-submission where communication between tasks uses files. These approaches typically use a cost/deadline resource management model. More recently, e.g. [11] and [12], there has been a trend towards using a just-in-time approach. Here, instead of analysing a workflow and trying to optimise a static schedule, resources are allocated on a first come, first served basis. The work described here differs from grid developments in a number of significant ways. Perhaps the first and most significant is that the *ÆTHER* project aims to build a complete programming solution to such distributed environments and in doing so, it has defined a model of concurrency that captures both work and resources in an abstract manner in a single integrated model [4]. We also adopt a just-in-time approach to scheduling but in our case this is required as the underlying SVP model is implemented at the level of instructions in a processor's ISA and adaptations to load may occur at MHz rates, giving little time for planning a schedule. Also note that this just in time approach adapts to situations where there may be a significant latency in setting up a remote resource to perform a computation. Two examples are just-in-time compilation for different instruction sets and device configuration in FPGA like devices. In *ÆTHER* there is considerable interest in the design of run-time support for reconfigurable SoCs [13].

7 Summary

This paper has presented the architecture of a hierarchical SANE system, where resources are shared between SANEs by delegating both work and the responsibility to meet the deadline or requirements for that work. This architecture builds upon the SVP model of concurrency that provides an abstraction of work as a family of threads and an abstraction of resources as a place. The protocol provides a place server to define the place at which the family of threads is executed once the protocol has been completed. The protocol proposed for negotiating the use of resources is based on a cost model that uses the required energy as a baseline cost, to be modulated by market forces. A baseline implementation could use cost as simply a selection criteria with no credits being exchanged at all. In this way threads could collectively minimise energy consumption in the system. With a cost model however much richer scenarios can be envisioned, where the cost, although based on energy, is dependent on market conditions, such that at periods of high demand cost would rise. In such a scenario, one can imagine, as with our financial world, a number of SANEs cornering the market on energy credits by speculating in the market. Such cost policies and mapping strategies will be evaluated within the remaining period of the *ÆTHER* project in order to understand their emergent behaviour.

8 Acknowledgements

The authors acknowledge support from the European Community in funding the research undertaken in the ÆTHER project.

References

1. Jan Krikke, "T-Engine: Japan's Ubiquitous Computing Architecture Is Ready for Prime Time," IEEE Pervasive Computing , vol. 04, no. 2, pp. 4-9, April-June, 2005.
2. C R Jesshope(2008) A model for the design and programming of multicores, to be published in: Advances in Parallel Computing, IOS Press, Amsterdam, <http://staff.science.uva.nl/~jesshope/Papers/Multicores.pdf>.
3. M. van Tol, C R Jesshope, M Lankamp and S Polstra (2008) An implementation of the SANE Virtual Processor using Posix threads, submitted to: Journal of Systems Architecture, <http://staff.science.uva.nl/~jesshope/Papers/Multicores.pdf>.
4. Jesshope C. R. (2006) Microthreading a model for distributed instruction-level concurrency, Parallel processing Letters, 16(2), pp209-228.
5. Bernard T A M, Jesshope C R, and Knijnenburg P M W (2007) Strategies for Compiling ?TC to Novel Chip Multiprocessors, International Symposium on Systems, Architectures, Modeling and Simulation, SAMOS 2007, S. Vassiliadis et al. (Eds.), LNCS 4599, pp.127-138.
6. Katarina Paulsson, Michael Hbner, Jrgen Becker, Jean-Marc Philippe, Christian Gamrat (2007) On-Line Routing of Reconfigurable Functions for Future Self-Adaptive Systems - Investigations within the AETHER Project, IEEE International Conference on Field Programmable Logic And Applications (FPL), pp.415-422, 27-29 Aug. 2007, Amsterdam, The Netherlands.
7. Smith RG (1980) The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, IEEE Trans. Comput., vol. C-29, no. 12
8. M.El Khodary, J-Ph.Diguet, G.Gogniat (2007) Operating Environment on-line Metrics for Application Architecture Matching, 25th IEEE Norchip Conf., pp19-20 Nov. 2007, Aalborg, Denmark.
9. Cao J, et al. (2003) WorkFlow Management for Grid Computing, Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 198-205.
10. Buyya R, Abramson D and Giddy J (2000) Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid, HPC Asia 2000, China.
11. Deelman E, et. al. (2004) Pegasus: Mapping Scientific Workflows onto the Grid. Across Grids Conference, Nicosia, Cyprus.
12. Omar W M, Taleb-Bendiab A and Karam, Y (2006) Autonomic Middleware Services for Just-In-Time Grid Services Provisioning, Journal of Computer Science, 6, pp 521-527, ISSN 1549-3636.
13. Marescaux T, et. al. (2004) Run-time support for heterogeneous multitasking on reconfigurable SoCs, Integration, the VLSI journal, 38, pp 107130.
14. J. H. Lepler (2004) *Cooperation and deviation in market-based resource allocation*, University of Cambridge Technical report, UCAM-CL-TR-622, ISSN 1476-2986.