Resurrecting Actors: New Applications of an Old Paradigm in Engineering and Science.

Tristan Aubrey-Jones <taj105@ecs.soton.ac.uk>

***Abstract*—This report reviews new research into an old programming paradigm for distributed computation called the Actor model. The model is described, and four exciting new applications are reviewed which apply the model to large scale distributed computations, wireless sensor networks, embedded systems, and highly concurrent programs. It is found that the actor model is being revived to provide an intuitive abstraction from the underlying implementation of these systems. It is thus being used by code generators and middleware platforms to allow programmers to produce portable and highly optimized applications for complex architectures, with far greater ease.**

I. Introduction

THE actor model [1] was proposed in the late 70's as an alternative to the shared memory model for concurrency, but was left stillborn. It is now being resurrected in a number of fields, to help programmers write efficient software for increasingly complex environments. By raising the level of abstraction, it enables code generators and intermediate platforms to produce adaptive optimized implementations, that would not otherwise be achievable.

This report reviews four recent contributions which all follow this pattern to help developers program large scale distributed computations, wireless sensor networks, highly concurrent programs, and complex signal processing systems (see Figure 1). In each case researchers have developed and subsequently evaluated a prototype system, and so each system will be briefly described, evaluated, and interesting results stated. Finally conclusions will be drawn about the success of the paradigm from all four.

II. The Actor Model

The actor model was proposed in 1977 [1] as an alternative model of concurrent computation to the concurrently proposed "communicating sequential processes" [2] and functional/data flow models. It differs most notably in that the model's computational elements called "Actors", both transform data values and maintain state, and that they can create new actors so systems are dynamic. As such it is more powerful than either of its contemporaries.

An actor is a computational element which maps an incoming communication to a triple: a new behavior/state, further communications to send, and new actors to create [3]. It can thus be likened to an object with a public mailbox, its own thread and entirely private state. Actors are naturally concurrent as every actor's state is isolated and can only be accessed externally via message passing. All coordination is via message passing and so there are no shared variables or locks, just patterns of passing messages. Thus actors can be co-located or communicate remotely without breaking the operational semantics of a program.

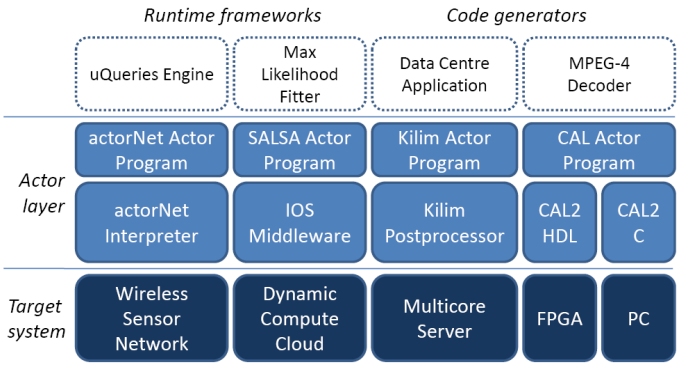

Figure 1. New applications of the actor model

This property can be useful in a number of situations.

For example researchers performing Computational Fluid Dynamics (CFD) simulations using many distributed computers, do not want to be concerned about the complexities of efficiently distributing the computation. CFD approximates the state of a fluid as an array of velocities, which are recomputed by solving equations for each cell at progressive time steps. A cell's value at $t + 1$ depends on its neighbor's values at t, and so the problem exhibits fine grained parallelism. However it can be made coarse grained by concurrently evaluating regions of the array and only sharing the edge values of each region [4]. This could be programmed using threads and a shared array with locks to police shared boundary values, but if we use actors the computation doesn't need shared memory and can be distributed over a network of machines. Each region can be contained as an actor, which sends messages to communicate its boundary values as per figure 2.

By making data dependencies explicit through message passing, the actor model introduces flexibility into how programs are realized and so they become highly portable, supporting many diverse platforms including heterogeneous distributed systems. This abstraction from the implementation makes it aptly suited for difficult domains, as essential program semantics can be captured concisely, allowing code generators and runtimes to decide on exact implementation details and include optimizations that would not otherwise be possible. However despite these advantages the Actor model remained largely unused for many years, due to the continual acceleration of single threaded hardware, the lack of truly distributed architectures, and the niche AI bias of the original research. It is only now with the increase in distributed architectures via wired and wireless networks, multicore systems, and code generation technology, that its usefulness is beginning to be realized.

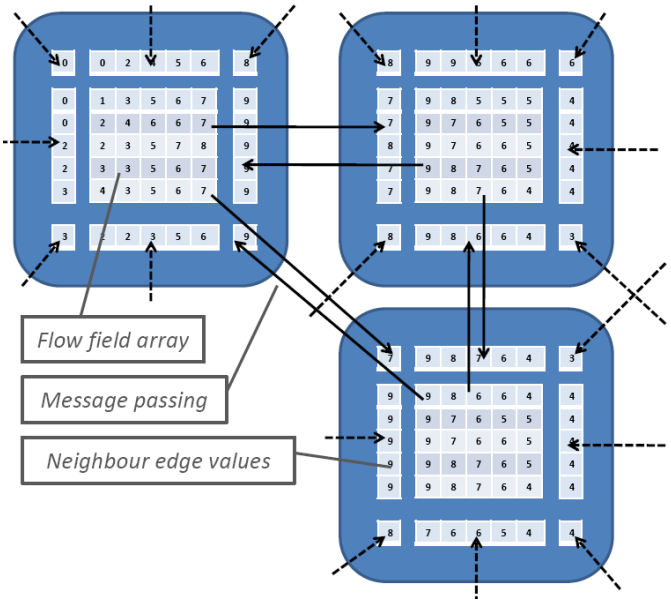

Figure 2. CFD Actor Communication

III. Large Distributed Computation: The Internet Operating System

With the increasing prevalence of multicore and distributed architectures, there is an increasing need for developer-friendly distributed computing frameworks. Existing application frameworks like MPI aid programming such systems, but it typically falls to the application developer to implement component migration logic and any reconfiguration/load balancing strategy. Dynamic load balancing can be complex to implement but can also have a dramatic effect on performance. This is especially true when the target network is heterogeneous, Internet scale, and dynamically changing. The actor model is ideally suited for this situation, as its isolated state and explicit message passing allows migration almost for free.

In 2001 the actor model was used as the basis of SALSA [5], an language for mobile and Internet computing built on top of Java. The language simplifies programming distributed systems by providing universal names, active objects and migration, such that it can be openly distributed and dynamically reconfigured. It allows the definition of "behaviors" like Java classes, containing internal variables and message handlers, from which many actors can be instantiated. Isolation is enforced by duplicating all messages before sending: a technique that can be very detrimental to performance. SALSA thereby allows programs to be written as systems of intercommunicating actors, which can be hosted locally, or distributed remotely.

This full potential of SALSA was realized in 2006 by the "Internet Operating System" [6], a middleware platform to automate dynamic reconfiguration of distributed actor systems and research various load balancing strategies. The IOS originally only supported SALSA but now provides an API for MPI. Unlike similar systems, IOS decentralizes all reconfiguration decisions, performing them at the nodes rather than at a central server. At each node there are three pluggable modules: a module which profiles application and system resource usage, a decision module which evaluates whether a specific reconfiguration is worthwhile, and a protocol module which sends "work stealing" requests to other agents and reconfiguration requests to the application. Load balancing occurs when newly joined or lightly loaded nodes, propagate requests to "steal work" until an overloaded actor is found, at which point it may be reconfigured and migrated to the underused node.

Various decision strategies have been evaluated on the system including random work stealing (RS) and "actor topology sensitive" random work stealing (ARS) which monitors the number of messages an actor sends to remote agents, in order to collocate tightly coupled actors. Experiments with different actor topologies have shown that although traditional RS yields lower message throughput's than round-robin (RR), by utilizing runtime knowledge about the actor topology, ARS can considerably exceed it, as it did by an order of magnitude on the massively parallel sparse benchmark [6]. In one experiment where nodes were added and removed from the network, the ARS version was running 4x faster at the end than the version that could not adapt. Recently the framework has been successfully used to implement a distributed maximum likelihood fitter for use in partial wave analysis of particle accelerator data using the simplex algorithm [7].

The research demonstrates that using IOS with ARS can dramatically improve performance, although has not yet been evaluated on any very large real world applications. The implementation of the maximum likelihood fitter shows the potential usefulness of IOS, but does not seem to have been evaluated on anything like an Internet scale network. However despite the lack of really large benchmarks the framework does show that treating the application as an actor system, not only eases migration, but allows dynamic profiling of communication patterns and efficient load balancing, without a priori knowledge about the application or target architecture. It is also interesting to note that the SALSA actor language does not seem to have been used in any real world application, until the IOS framework implemented some of the optimizations that it theoretically made possible.

IV. Wireless Sensor Networks: ActorNet

Another topical hard-to-program environment is the wireless sensor network, where small wirelessly communicating devices need to coordinate and collaborate to harvest data or perform a task. These distributed systems underpin the emerging "Ambient Intelligence" and can be even more difficult to program than large scale systems due to their primitive operating systems, memory limitations, need to conserve energy, and inability to ignore spatial distribution. For this reason various programming techniques are being researched to raise the level of abstraction such that WSNs can be programmed with more powerful constructs. A number of "macro-programming languages" have been researched which translate a global

system behavior into individual sensor programs, but one promising new approach called "actorNet" [8] goes further by using the actor model to develop a mobile agent platform for WSNs on top of which domain specific macroprogramming systems can be built.

The core of actorNet is a variant of Scheme augmented with actor primitives and a fine-tuned, multithreaded interpreter written to run on TinyOS and the Mica2 hardware. This decouples programs from the sensor OS (aiding portability), and allows them to make use of virtual memory, garbage collection, and efficient blocking IO via an application-level context switching service. In addition to these benefits the actor extensions enable multitasking (par), asynchronous message passing, and actor migration such that actors can roam and replicate across the network as mobile agents, and sensors can be remotely reprogrammed. A key aim for WSNs is to perform processing at the sensor, in order to conserve energy by reducing communication, and so these "mobile actors" provide an ideal execution environment as they enable resource aware task allocation, i.e., for detection code to be moved to the sensor. This system therefore provides the necessary foundation for additional services like in network storage, and more powerful macroprogramming.

In fact the first query-based macro-programming language called "uQueries" uses actorNet to provide the combination of dynamicity and query specification useful to domain experts [9]. uQueries are translated into "meta-actors" that spawn mobile actors which migrate across the WSN to perform the query. Knowledge representation techniques are used to track domain knowledge and enable various query processing adaptations including automatic optimizations to minimize energy usage.

The dynamicity and migration abilities of actorNet make it well suited to flexible coordination of WSNs, although it remains to be seen how useful it will prove in real world applications. Abstracting away from the operating system greatly improves portability, however the performance penalties do not seem to have been evaluated. Future work should compare the system's execution speed, communication overhead, and energy usage to other systems. Should the system's efficiency be acceptable, it could well become a very successful paradigm for programming WSNs, although this will partly depend on the success of derivatives like uQueries which exploit the model's mobile agent capabilities.

V. Concurrency-Oriented Programming: Kilim

Physically distributed systems aren't the only situation actors are being used to tackle. They are also being used as a concurrency primitive for data centre applications with split phase workloads, where many concurrent tasks need to share some resources and so efficient and reliable concurrency constructs are essential. It is difficult to obtain correctness, fairness, and efficiency using shared objects and fine-grained locks, and so various "concurrency-oriented languages" like Erlang have been developed which use actors and message passing instead.

In 2008 a new framework called "Kilim" was presented to help develop "robust massively concurrent systems in mainstream languages" which uses a code post processor to augment Java with ultra lightweight cooperatively scheduled threads (actors) and isolation aware message passing [10]. It thereby maintains all the familiarity of object oriented programming whilst offering the robustness of isolated concurrent threads with no locks and no shared memory. Kilim actors are essentially Java classes which extend `Task` and provide a public `execute` method marked with the `@pausable` annotation as per the example below.

```
class HttpConn extends Task {
    @pausable public void execute() {
        while (true) { HttpMsg m = readReq();
            processMsg(m); }
    }

    @pausable public HttpMsg readReq() { .... }
}
new HttpConn(mbox).start();
```

Scheduling is optimized by mapping many actors onto a few threads through the application of a continuation-passing-style transformation on all `@pausable` methods. Whenever `Actor.pause()` is called, the actor cooperatively stores its current continuation, and the scheduler resumes another actor from its continuation. This is implemented efficiently by weaving an extra `fiber` parameter into all `@pausable` methods to signal to callers when an actor has paused, and provide a store for its activation frame. This is further optimized by detecting duplicate values and constants during heap analysis, and omitting them from the `fiber`. This task switching mechanism is 1000x faster than Java threads, and 60x faster than other lightweight task switching frameworks.

In most actor languages state isolation is enforced by cloning all messages before sending [5] causing a significant performance penalty. In Kilim this is overcome through the use of a "linear type system" whereby messages can only be owned by one actor at a time, being destructively read when sent, such that messages can be passed by-reference. In this system messages must be unencapsulated values with no internal aliasing, and therefore form public tree structures. Isolation is enforced statically by determining the capability of message objects at every node in the control flow graph using heap analysis. A message may either be *free* (can be assigned to other messages and sent), *cuttable* (can only be sent or assigned if destructively read), *safe* (cannot be modified or sent), or *invalid* (once sent or destructively read). Annotations are attached to method parameters to require a certain capability is provided by callers and thus guaranteed to the callee. For example in the method `foo` below, `p` can only be assigned to a non-safe variable, and once it has been assigned cannot be aliased but only destructively read, such that it is only ever part of one message.

```
void foo(@free Event ev, @safe Event msg) {
```

```
    p = new Event(); // p & ev are free
    msg.a = p; // error: msg is safe
    ev.a = p;  // p becomes cuttable
    ev.b[2] = p; // error: p is not free
    ev.a = msg; // error: msg is not free
    mailbox.send(ev); // ev becomes invalid
}
```

Micro-benchmarks have shown that Kilim is considerably faster than Erlang, the current standard for concurrency-oriented programming, performing 3x faster at message passing, and 4x faster at actor creation, although real world applications are still needed to evaluate its fairness, cache locality and memory usage. More importantly though Kilim is a promising step towards efficient and robust actor based concurrency in main stream object oriented programming. The statically enforced memory isolation simplifies reasoning about concurrency, whilst the linear ownership type system [11] enables efficient transfer of messages and shared resources (using the `@sharable` annotation). However the system is not yet complete: further research is needed to formally prove the correctness of the type system, the fairness of the system needs evaluating, and its usefulness in a real world application still needs to be demonstrated.

VI. Complex Signal Processing Systems: CAL

Actors are also beginning to be used to assist the development of signal processing systems. Signal processing systems are becoming increasingly complex, such that efficient design can no-longer rely on intuition, and tools that unify architectural and algorithmic design are required [12]. Current C++ based systems lack the concurrency operators required by embedded systems, and require an unreliable multiphase design process to transition from architecture to implementation. Here again actors are providing a more suitable programming abstraction, decoupling design from implementation, and automating implementation and optimization through automatic code generation.

In 2003 the CAL language was defined by the Ptolomey research group at UC Berkley [13]. CAL is an actor/dataflow oriented language, which defines a high level abstraction for composing static networks of concurrently executing actors which communicate via token passing through well defined ports. Unlike the original definition of the actor model actors cannot instantiate more actors enforcing a static topology. This constraint is implicit in embedded architectures and enables diagrammatic representation and visual programming to greatly ease design. Furthermore hierarchical design is possible as actors can be composed of networks of other actors, such that entire algorithms can be designed graphically. The language therefore greatly improves the understandability of a design, and its greater descriptiveness improves conciseness. For example an MPEG-4 decoder system required only 4000 lines of CAL compared to 15000 lines of VHDL and 4100 of optimized C++ [12]. It is not surprising then that the ISO/IES have chosen to use CAL in the new MPEG "RCV" standard (ISO/IES 23001-4 & 23002-4).

In 2008 researchers presented a framework using CAL to model, simulate and implement signal processing systems [12]. The framework contained 2 code generators: CAL2C and CAL2HDL which produce a C program or a VHDL description respectively, from a CAL model. Benchmarks implementing an MPEG-4 Simple Profile Decoder showed firstly than the C code was 130x faster than the simulator, but more importantly that the VHDL generated was almost 4x smaller, performed 1.6x faster, and was reportedly 4x faster to develop than an existing manually coded VHDL implementation. It is possible that other models would not perform this well, especially as the framework doesn't currently perform any cross-actor optimization, however recent research claims to address this by using "partial evaluation" as an optimized compilation technique to yield (almost) no performance penalty [14].

CAL's static topology does limit the system's wider applicability, but for its intended domain, significantly increases its usefulness. The MPEG-4 example demonstrates CAL's usefulness and benefits in a real world application, as the system effectively exploits the actor programming paradigm.

VII. Conclusions

The independent adoption of the actor model in four separate research areas indicates that the actor model is being revived. When programmers can efficiently reason about system behavior by intuition there is no need for tools or paradigms to assist them, but when it becomes impractical or impossible for intuition to produce an efficient system, tools are needed to derive an implementation from a high level abstraction. But now, as target architectures become increasingly complex, more flexible abstractions and programming tools to exploit them are becoming a necessity, and so the usefulness of actors are beginning to be demonstrated.

As Figure 1 illustrates the research reviewed indicates that as code generators and middleware platforms are becoming necessary, the actor model is being used to provide a new level of abstraction. This helps make programming complex systems intuitive, and provides the architecture independence required to be able to automatically generate efficient implementations for different platforms. In some situations, like with actorNet, that abstraction may then support more powerful domain specific languages like uQueries that would not be possible without the actor layer. All four contributions adjust the model slightly to suit the domain, but the common themes remain. Kilim is particularly interesting as it introduces actors with linear types, showing them to be a very useful addition to the actor model, allowing resources to be shared between actors (by passing them as messages), and providing scope for efficient local zero-copy message transfer.

In general the research reviewed agrees that the shared memory assumption is an unhelpful one, limiting the portability and flexibility of systems. The actor model on the other hand, is providing a common paradigm for all

four systems, simplifying the development of distributed applications of all sorts, and allowing optimizations that would be impossible in the former. It is early days for all four applications, but it seems the predicted benefits of actors are finally beginning to be realized over 30 years after their debut.

References

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," Massachusetts Institute of Technology, Tech. Rep., 1976.

[2] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–667, 1978.

[3] G. A. Agha, *Actors: A model of Concurrent Computation in Distributed Systems*. The MIT Press, London, UK, 1986.

[4] S. Barnard, R. Biswas, S. Saini, R. V. der Wijngaart, M. Yarrow, and L. Zechtzer, "Large-scale distributed computational fluid dynamics on the information power grid using globus," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, p. 60.

[5] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, vol. 36, no. 12, pp. 20–34, Dec. 2001, http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[6] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela, "The internet operating system: Middleware for adaptive distributed computing," in *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 2006, p. 2006.

[7] W. jen Wang, K. El, M. John, and C. J. Napolitano, "A middleware framework for maximum likelihood evaluation over dynamic grids," in *In Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[8] K. M. YoungMin Kwon, Sameer Sundresh and G. Agha, "Actornet: An actor platform for wireless sensor networks," 2006.

[9] R. Razavi, "Dynamic macroprogramming of wireless sensor networks with mobile agents," in *In 2nd Workshop on Artificial Intelligence Techniques for Ambient Intelligence*, 2007, pp. 43–55.

[10] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java (a million actors, safe zero-copy communication)," 2008.

[11] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*. ACM Press, 1998, pp. 48–64.

[12] C. L. et al., "Dataflow/actor-oriented language for the design of complex signal processing systems," in *In Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP 2008), Bruxelles: Belgique (2008)*, 2008.

[13] J. Eker and J. W. Janneck, "Cal language report: Specification of the cal actor language," University of California at Berkeley, Tech. Rep., 2003.

[14] M.-K. L. Gang Zhou and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," in *In Proceedings of International Conference on Embedded Software and Systems 2007*. Springer-Verlang, 2006, pp. 786–799.