Multi-agent Platforms and Asynchronous Message Passing: Frameworks Overview

Christopher Frantz
Mariusz Nowostawski
Martin Purvis

The Information Science Discussion Paper Series

Number 2010/07
November 2010
ISSN 1177-455X

Multi-agent platforms and Asynchronous Message Passing

Frameworks Overview

Christopher Frantz Mariusz Nowostawski Martin Purvis

{cfrantz,mnowostawski,mpurvis}@infoscience.otago.ac.nz
Information Science Department
University of Otago
Dunedin, New Zealand

ABSTRACT

In this article we review contemporary multi-agent system architectures and implementations. We particularly focus on asynchronous message passing mechanisms. Our motivation is to explore two main areas in the context of multi-agent systems: the concept of micro-agents and the asynchronous message passing architectures. In the article we take a close look at the emerging area of micro-agent-based systems and contrast them with selected representatives from the general field of agent architectures. We provide historical references and examples of contemporary implementations supporting the hierarchical micro-agent-based software engineering paradigm. In addition, we also investigate various implementation mechanisms for efficient asynchronous message passing between large numbers of small interacting software components with regards to their use in the context of multi-agent systems. The results show a trade-off between performance, fairness and usability as key problem when selecting an appropriate solution. Future investigations into alternative concurrency handling mechanisms for better support of micro-agent architectures are suggested.

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation and Modelling; I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence

General Terms

Software Architectures

Keywords

multi-agent systems, micro-agents, asynchronous, communication, message passing

1. MOTIVATION

The field of Multi-Agent Systems (MAS) is driven by the key metaphor of interaction between multiple autonomous agents whose micro-behaviour produces the perceived overall system behaviour. Given this broad understanding, it is unavoidable that the field of MAS is all but unified. It is rather dispersed into various application-related categories e.g. social sciences, education, large-scale distributed systems and so on. Independent developments in those different subfields drive this diversification further, limiting the perspective for a more precise general concensus on fundamental concepts and their development. Standards encouraging the development of open general-purpose platforms, carefully respecting key agent properties, have been specified – e.g. Foundation for Intelligent Physical Agents (FIPA) specifications [16], Java Agent Services (JAS) [4] for a unified platform implementation in Java. However, efforts to implement those standards effectively decline[1]. A successful example of this group of standard-compliant general purpose platforms is the Java Agent DEvelopment Framework (JADE) [9] which is still under active development and used in various applications and organizations (see [12]).

It is our belief that the lack of efficient MAS platforms is the inhibiting aspect of the multi-agent paradigm. We believe that although existing frameworks promote the main open system principles, they are nevertheless not particularly suitable to represent flexible agent entities and interaction on various levels as necessary for composition and organization of higher-level entities in the spirit of true multi-level Agent-Oriented Software Engineering (AOSE) [21]. Reason for this is the overhead of FIPA-agents, enforced by standard-compliance such as the global naming scheme, high level of abstraction from message transport mechanisms to support the various transport protocols (such as IIOP, HTTP and SMTP) and last, but not least, the assumption of a rational agent architecture imposed by its knowledge-level communication language (see [26] for a detailed discussion).

As such the work presented in this paper deals with several areas yielding towards a consistent AOSE approach while providing high runtime performance. Micro-agents, albeit following the intentional stance, shall be perceived as

[1]Most of the platforms listed under [8] ceased to exist or have not received further development for the last few years; today JAS is an inactive Sourceforge project.

continuous concepts with a streamlined set of core features and a flexible demand-oriented extension filling the gap below FIPA-compliant agents. In the trade-off of performance and expressiveness micro-agents compromise on the latter and act as a complement to the conceptually more powerful FIPA-agents which are eventually the result of micro-agent composition. This allows the consequent use of agent-related concepts on every level without sacrificing overall system performance.

Additionally, the micro-agent paradigm respects the diversified agent understanding throughout the whole community (and especially application-related subfields) and encourages the consistent use of concepts and constructs, avoiding the confusion with related development paradigms like object-orientation. Although complexity of the concepts might be similar at the lower levels we perceive the blended use of both paradigms as one of the pitfalls of AOSE.

Key aspects to achieve this is to firstly, provide an efficient underlying asynchronous message passing mechanism focusing on performance and scalability and, secondly, relieve the developer from the underlying threading aspects which should naturally be managed by the agent infrastructure rather than the implementer.

In the first part of the article we introduce our micro-agent architecture. We provide the necessary conceptual framework and link the concept of micro-agents to existing developments employing hierarchical agent organizations which have been undertaken in the field of MAS.

Following this we provide an overview of selected Java-based asynchronous message passing frameworks and investigate their applicability in the context of multi-agent systems. We highlight the key aspects of usability, fairness and performance. We concentrate on Java-based frameworks as of the broad adoption of this language for the implementation of multi-agent platforms.

2. MICRO-AGENTS

The MAS standardization efforts driven in the 1990's and early 2000's were majorly concerned to see agents as enablers for truly open systems engaging in *intelligent* conversations. In consequence, the modelling of practical applications frequently demanded the decision whether to use heavy-weight agents, switch between object-oriented and agent-oriented development or fall back to the individual development of agent platforms completely breaking with the vision of open agent-based systems. To bridge the conceptual gap between the high-level agent-based abstractions, typical programming frameworks and data structures (typically object-oriented or structured programming), we have proposed the notion of micro-agents [24]. Micro-agents fill the conceptual and implementation gap between the modelling abstractions of agent-hood, and implementation-oriented constructs used to actually implement the modelling abstractions. Micro-agents are designed for providing performance while supporting a scalable agent notion. They provide a much better support for various software engineering demands rather than following the *one size fits all* approach, taken by most popular agent development frameworks. We do not argue that those systems are unsuitable in general; we rather argue that the efficiency penalties involved in using those systems outweigh benefits from the use of agent-oriented software engineering for production environments and create the general perception of agents to be difficult to work with and slow.

Indicators that the existence of this kind of agents is useful come from the field of agent-based simulations which heavily rely on the concept of emergence, enabled by considerably large number of simple agents. Although the latter notion is often too simplistic and implementations hardly offer direct communication (see Malsch et al [23]), multi-agent systems can learn from this and extend this to a rather continuous agent notion, the concept of micro-agents.

Multi-agent platforms known to have micro-agent-like entities are currently limited to two general-purpose systems, namely MadKit and Opal. Both of them will be introduced in this section, along with other more popular toolkits such as JADE and 3APL.

2.1 MadKit

The MadKit multi-agent platform [18] has been developed at the University of Montpellier in the late 1990's and offers a lean agent concept based on the Agent-Group-Role (AGR) model which does not enforce a specific agent architecture. In this a general agent is simply considered as " ... a class defining [a] basic life-cycle ... " [14] which is backed by a micro-kernel including group management capabilities, the life-cycle management as well as the message passing mechanism. Given those constraints the basic unit of modeling is in fact an organisation consisting of agents which are associated with (potentially multiple) groups and play roles. This allows the 'cheap' use of agents - an effect which the platform uses itself: All services, apart from the one associated with the micro-kernel are so-called *agentified services*.[2] Groups of agents communicate within their group. Communication to external parties is always established via one agent playing the role of the *communicator*. All further specific services, including communication services are implemented as specialized agents which then can handle a specific message protocol for the whole platform (e.g. socket-based cross-platform communication).

The obvious goal of the platform is to achieve a useful basis for various kinds of multi-agent systems and rather than providing a fully-fledged featureful system concentrates on an efficient core which allows extension in a consequently agent-oriented manner. The development of agents is eased by providing explicit support for the JESS rule language, Scheme bindings as well as pre-defined models for various purposes (e.g. simulation of artificial life). [3]

2.2 JAS

The *Java Agent Services* (JAS), driven by the Java Community Process and resulting in Java Specification Request 87 (JSR87), has the goal to specify unified interfaces for the implementation of the FIPA Abstract Architecture for Java-based FIPA-compliant agent platforms. Although the FIPA specifications are detailed on the semantic level, the platform-independent nature leaves a strong degree of engineering freedom with regards to the implementation. JAS aims at closing this gap by providing the according Java interfaces agent platform implementation could eventually commit to. As such its listing under the agent platform is not entirely correct but it represents the contribution to Java

[2]Additionally generic graphical components allow reuse for GUI development.

[3]The version considered in this review is 4.2.0.

itself with regards to the standardization efforts of agent platforms.

Its value is to abstract an platform implementation from lower-level transport mechanisms and unification of FIPA directory registration services (agent naming, de/registration) to not only ensure interoperability between different platforms but also to keep a certain degree of technology-independence and allow reuse of transport services in different platforms. The specification version 2.1 dates back to 2002 but despite the promising concept it did not gain strong attention for the implementation in agent platforms. Platforms known to still make use of JAS are OPAL (in an extended version) as well as the reference implementation of 3APL[1].

2.3 Opal

The Otago Agent Platform (OPAL) has been developed at the University of Otago from 1999 onwards, building on a simple micro-agent kernel and continuously extended towards a fully-fledged FIPA-compliant agent platform, ranging from a weak micro-agent notion to potentially rational high-level OPAL agents. Reason for this is not only the support for different application types but also the consequent composition of high-level agents from micro-agents. The multi-level architecture behind this concept is defined by its three levels of agents and represents an extension to the multi-level architecture proposed in [24].

Primitive micro-agents live on the lowest level and provide capabilities similar to objects, including a purely reactive, if not only responsive, internal architecture as well as direct method invocation. In contrast to objects, the discovery and linking mechanism is dynamic and goal-driven. Due to their primitive nature these agents do not compromise performance.

Non-primitive micro-agents constitute the next level of abstraction and are the key aspect of this work. They provide an organizational scheme for primitive micro-agents, make use of primitive micro-agents for their own goal achievement, rely on asynchronous message passing as interaction mechanism and manage their own logical thread of execution. Additionally, they can be implemented in programming languages other than Java (e.g. Clojure) and can transparently engage in internode communication. The use of non-primitive micro-agents involves some limited performance penalty but provides a significantly bigger feature set. In order to minimize the memory footprint lazy initialization is used as key principle throughout the micro-agent platform as well as its agents and includes messaging, organization mechanisms, programming language support and networking. Taking this into account, capabilities and 'weight' of non-primitive micro-agents vary in an application-related manner, supporting both applications with a limited number of seemingly intelligent entities (e.g. auctions) as well as hundreds of light-weight entities (e.g. ants).

On the highest level we advocate system openness and emphasize standards compliance for the agent entities which concerns FIPA standards as well as the adoption of an extended version of JAS. On this level the system is consequently FIPA-oriented and, among other specifications, supports the Abstract Architecture and Agent Communication Language (ACL). The implementation additionally complies to an extended version of JAS to allow a strong abstraction from the message transport layer. Considering the fact that higher-level agents necessarily inherit capabilities from lower levels by extending them; moving to a different level does not result in a loss of capabilities. As such the proposed system harmonizes the trade-off between development of high-performance platform-specific systems as well as systems able to engage in communication with third-party platforms.

2.4 JADE

The Java Agent DEvelopment Framework (JADE) [13] [9] is probably the most prominent agent development framework written in the Java programming language, developed by Telecom Italia from 1998 onwards. The platform fully complies with the FIPA standards for the Abstract Architecture and the ACL as well as interaction protocols. In JADE agents are modelled as individual threads comprising their own behaviour and live in so-called containers – which do not necessarily need to reside on the same node. A platform consists of one main container and further containers holding the actual agents. The main/ *bootstrap container*, holds directories about all running containers and the according agents. The platform supports the development of fault-tolerant applications and provides support for mobile agents. The platform additionally allows the interaction with web services and is available as light-weight version for the use on mobile devices running Java 2 Micro Edition (J2ME). Although JADE does not focus on a fixed concept of agent-hood itself, the consequent implementation of the FIPA specifications forces it to implement a considerable conceptual baggage which makes the platform rather "heavy". Means of structuring agents, apart from their association to containers, are not considered.[4]

2.5 3APL

3APL [1], or more concrete *An Abstract Agent Programming Language*, has been developed at the University of Utrecht, Netherlands, to provide a full specification as well as reference implementation and development environment for an agent programming language. Although the actual language, like many other agent programming languages, is based on PROLOG, its major architectural difference to the popular AgentSpeak is the introduction of the plan revision cycle, allowing the revision of selected plans during execution.

3APL strictly focuses on rational agents; the developer needs to consequently adopt the 3APL agent model to build agents. As a result the computionally expensive execution makes the use of 3APL reasonable for applications with a limited number of intelligent agents (e.g. card games). In order to make use of external Java functionality, the platform allows the development of plugins to encapsulate Java code.

The messages are considered to be FIPA-compatible and the platform makes use of JAS. However, the message constructs in fact only provide a subset of the FIPA message specification. Communication between 3APL platforms is provided in a client-server pattern. As of its character as reference implementation the IDE puts major focus on capabilities to retrace the reasoning cycles and message logging, rather than providing a robust and efficient runtime platform. In the context of this comparison, the agent concept of 3APL is the by far most sophisticated but also most

[4]The JADE version considered here is 4.0.1.

constrained one and does not consider weaker agent notions. A simpler, more practical, approach to agent development is provided with 2APL, *a practical agent programming language*, which builds on top of the JADE platform. The authors also provide a mobile version of 3APL, 3APL-M targeting devices running J2ME.[5]

2.6 Performance comparisons

The platforms listed above support a range of different agent notions and platform capabilities. To retrace the differences of those platforms and to facilitated the proper choice of the most suitable platform for an application, we have conducted a performance comparison based on a simple agent scenario focusing mainly on agent interactions.

The scenario involves four agents: A client agent requests a service (represented as a data retrieval and print on the console) from another agent which itself coordinates the data retrieval and print via two agents dedicated to the according actions. The activity is repeatedly requested by the client until a given number of rounds is reached. Then timer is stopped. Although the agent functionality required to conduct the tasks is simple, it involves the coordination of interaction, especially by the mediating agent.[6]

The results (see figure 1 and table 1) show three clusters of platforms. The certainly by far slowest is 3APL with its extensive reasoning, largely interpreted execution, but also the overhead caused by its logging facilities.

	JADE	3APL	MadKit	OPAL	Micro-agents
1000	0.25	1422	0.125	1.1	0.08
10000	8.3	-	1.025	10.4	0.41
100000	888.2	-	9.8	100.5	4.5
1000000	-	-	96	1035.7	42.2

Table 1: Benchmark results for Agent Platforms per scenario rounds (in seconds)

JADE agents and OPAL agents shape another cluster of heavy-weight platforms with considerably long runtime. In the test, despite assigning 1024 MB heap memory, JADE was not able to execute the scenario for 1000000 rounds. Although OPAL could achieve this task, it took fairly long. MadKit and the micro-agents of the OPAL platform outperformed any other platform. The OPAL micro-agents are about twice as fast as MadKit's agents. Especially for small but high-performance tasks micro-agent frameworks seem a useful alternative to heavy-weight platforms while keeping their flexibility for extension with further capabilities (e.g. different implementation languages, reasoning engines).

3. ASYNCHRONOUS MESSAGING FRAMEWORKS

Asynchronous message passing – popularized in the context of the Actor model (see [20] and [10]) for concurrent computation and used as key communication paradigm in programming languages originally focused on inter-node communication such as Erlang [11]. The idea is based on the idea to fully decouple communicating entities and control structures and thus allowing non-blocking communication while following the *share nothing* principle. Its scalability potential by avoidance of empty wait cycles has received attention as viable alternative to shared memory approaches for interprocess communication in the context of multi-core computing and its inevitable consideration in contemporary programming [28].

In this context, independent from the technological trends, asynchronous communication and the maintenance of several concurrent conversations is very natural for agents. In turnaround the increasing availability of asynchronous message passing frameworks motivates a review with regards to their suitability as a messaging infrastructure for agent-based systems. Although most of the reviewed frameworks see their use in the context of the actor pattern, many do not fulfill key semantic properties for actor systems (encapsulation of state and messages, fairness, location transparent addressing and mobility) [22]. In this context we trade some of those properties against the importance of performance as we see the frameworks as infrastructural basis for agent platforms which eventually balance unsatisfied properties.

Alternative mechanisms for interprocess communication using the message passing principle include *Remote Procedure Call* (RPC) which, introduced around 1976 [30], allows the transparent integration of calls to procedures - or methods in the context of object-orientation - on remote nodes into local code. Due to the transparent integration blocking execution on caller side is a key characteristic along with the necessary handling of network failure (which limits the usefulness of transparency). Further the RPC mechanism resulted in various incompatible RPC protocols and implementations. Examples include, but are not limited to, Sun's Remote Method Invocation (RMI), Microsoft's MSRPC or as part of the Common Object Request Broker Architecture (CORBA). As of the lack of message composition and aggregation features, frequent use results in decreased code performance and considerable network load and was one of the key drivers for the development of mobile agents, e.g. Telescript [29].

Web services [7] provide another interaction approach. Web services rely on W3C standards and have evolved in three different flavours, the first resembling the RPC pattern with open standards but providing similar drawbacks as of the poor abstraction from implementation details and as such comparatively tight coupling. The second type, established in the context of service-oriented architectures, concentrates on messages rather than operations described by comprehensive Web Service Description Language (WSDL) contracts. This results in less tight coupling but often heavily relies on developer introspection harming the ad-hoc use. Representational State Transfer (REST) web services switch the perspective towards a fixed set of operation primitives for interaction with web services and thus easing the dynamic invocation of their encapsulated (stateful) resources [15]. Although the used protocols promote openness a drawback is their dependence on HTTP as message transport protocol and as such comparatively poor performance.

CORBA [17], driven by the object-oriented programming paradigm and maintained by the Object Management Group (OMG), seeks to provide an access mechanism to share ob-

[5]The version considered for this review is dated 19 November 2007.

[6]The benchmark was undertaken on an Intel Core2 Quad-Core PC at 2.66 GHz, 3.25 GB RAM, using J2SE 1.6 Update 20 on Microsoft Windows XP Professional SP3.

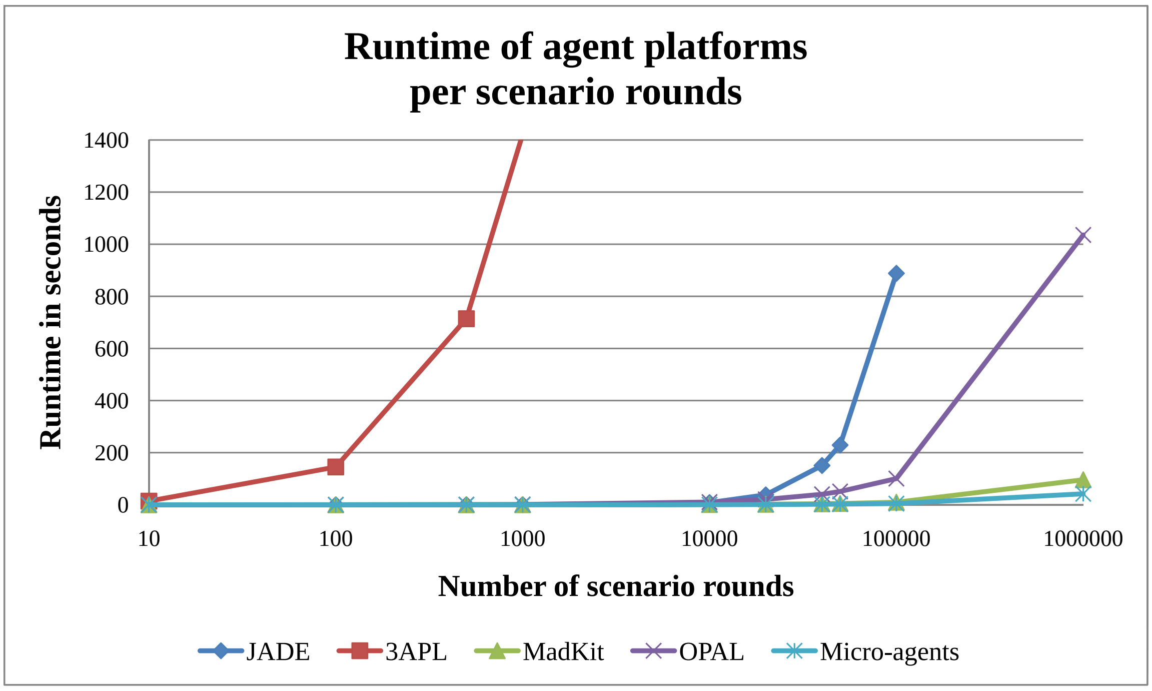

Figure 1: Performance benchmark results Asynchronous Messaging Frameworks

jects independent from programming language and locality. Each node uses an individual object request broker (ORB) through which it accesses objects by unique addressing. Mappings for concepts of the according programming language need to be existent in order to map object interfaces, described by the interface description language (IDL). CORBA provides a comprehensive set of specifications allowing various kinds of communication, including transport via HTTP. However, its use over the internet has been constrained as firewalls had been insufficiently considered. Along with this different implementations proved incompatible and threading was hardly taken into account during its specification [19].

In the following we review several existing Java-based message passing frameworks with regards to general functionality as well as performance in an agent-related scenario.

3.1 Kilim

Kilim [27] praises itself as an ultra-light-weight message passing framework which relates to the actor pattern and eliminates any user-code locking. Arbitrary numbers of light-weight cooperative threads interact via mailboxes. The light-weight threads are considered 'as light' as Erlang's fibers. Mailboxes wrap buffered typed message queues (using Java Generics) and can be shared and passed around in messages themselves. Features include the support for blocking, non-blocking, timed-blocking or selective retrieval from multiple mailboxes. Additionally the sizes of message queues can be bounded and support priorities. Messages are isolated in the sense of interference freedom; references to them can only be held by one entity at a time.

The development process involves an extended tool chain as a mandatory *Weaving* process amends annotated methods from the bytecode in order to produce light-weight cooperative threads. Kilim provides a central scheduler to ensure basic FIFO execution of tasks, additionally network support is provided using Java Asynchronous I/O. Kilim is released under an MIT-like license.[7]

3.2 Jetlang

Jetlang [5] is an open source 'high-performance' in-memory message passing library for Java developed by Mike Rettig and is based upon its .NET sibling retlang which resembles Erlang's implementation of asynchronous communication mechanisms for object-oriented programming languages. Jetlang adopts the Erlang-like notion of fibers and introduces typed channels as shared thread-safe entities using the Publish-Subscribe principle (supporting multiple subscribers) resulting in an extremely loose coupling. It further provides the concept of event subscriptions. Jetlang does not rely on a central scheduler; it only guarantees ordered message delivery to particular fibers in best-effort manner. Jetlang can be used in conjunction with different JVM languages such as Scala, Groovy or Clojure. As of its focus on in-memory message passing, networking is not available out-of-the-box. Jetlang relies on Java 1.6 and is available under [5][8].

3.3 ActorFoundry

ActorFoundry [2] belongs to the older group of actor frameworks for the JVM and has been developed at the Open Systems Lab at the University of Illinois. Its primary design goal is to strictly resemble the actor semantics and relieve developers from any underlying message handling. Criteria it seeks to fulfill are actor state encapsulation, safe message-passing (avoiding zero-copy messaging respectively

[7]The tested version (0.7.2 at time of writing) was dated 7th April 2010.

[8]Its current version 0.2.1 is dated 5th April 2010.

'message-by-reference'), fair scheduling, location transparency[9] by using a globally valid naming scheme and mobility [22]. Its original implementation mapped each actor on an individual thread and realized message transport using Java's serialization mechanism. In order to achieve performance improvements by avoidance of context switching ActorFoundry switched to the use of the cooperative light-weight threads used in Kilim (to avoid 1:1 assignment between thread and actor) - and as such inherits the post-compilation weaving step. Additionally a new scheduler has been provided.

ActorFoundry is not available open source; the "documentation by example" approach hampers the full functional understanding. Its current version 1.0 can be obtained under [2][10].

3.4 Actors Guild Framework

The *Actors Guild Framework* [3] yields at making thread programming in Java easier and indeed provides a light framework which makes heavy use Java annotations (as done in earlier versions of Kilim) to indicate messages and generate properties. Similar to ActorFoundry message types are defined by annotating Java methods which then can be accessed by other actor entities. Actors Guild intercepts accesses to those methods and queues them at runtime. Mutable data structures passed to methods (or messages in this context) need to implement Java's Serializable interface, similar to ActorFoundry.

Similarly to the latter the framework is less focused on performance but easy development and safe execution. This is partially done by relying on immutable data types (by modifying Java types to be finalized). It allows parameter manipulation from user code (thread pools, consideration for I/O tasks) and supports 'safe' (execution order sequential) and 'less safe' (execution order unpredictable) execution modes. Interestingly and different from any other framework it identifies an agent as a higher level entity as compared to its actors.

There is no indication in the source code that network support was considered. The framework is provided open source but does not report to have an existing community (the author is demanding for feedback on his framework) and development will eventually cease (as shown on the website). One reason for this is probably the poor performance, as shown in third-party benchmarks (see [25]). Compared to the other frameworks the documentation is extensive. The latest version of the framework can be found under [3] [11].

3.5 Korus

Korus is another framework promising to be light-weight and considers itself to be one ".. of [the] first 'pure' actor pattern frameworks in Java." [6]. Apart from all similar features provided by other frameworks it appears to have the simplest API of all frameworks seen before. Along with this it provides a distributed mode which allows network communication. This communication is not only restricted to communication with nodes of its own kind but also other frameworks or platforms integrated via adapters. An example adapter for communication with Erlang nodes is available at given time. Other than the frameworks mentioned before it includes a simple directory allowing lookup of other 'processes' (as used in the Korus terminology). Along with the network adapter functionality it anticipates the development of add-ons and its use for web applications by considering priorities for web requests. The Korus API supports developers in writing own parallel and pipeline execution constructs. Key difference to other frameworks is the fixed message type which essentially is a hashmap restricting key and value to String objects, obviously mainly to realize compatibility to third-party systems when considering distributed execution. Its latest version can be found under [6][12].

3.6 Messaging summary

Common for nearly all open source Java actor libraries is the limited community around them. Also, implementations supporting distributed communication are rare. Most frameworks gain their high performance by explicitly neglecting safe message passing, another relevant criteria disqualifying many of the frameworks to be associated with the actor pattern.

Application of both Kilim and ActorFoundry is majorly characterized by the necessary post-compilation for which no IDE support is available as of now. Additional drawback is the often limited informative value of error messages produced during the weaving process. In contrast, using Kilim's mailboxes principle is realized in a very intuitive and productive way. ActorFoundry repeatedly quit its service without any notification; it's intransparent nature (as non-open source software) can thus lead to extensive debugging efforts. ActorsGuild, although the by far worst performing framework used a similar approach to ActorFoundry by annotating methods as messages. However, the extensive use of annotations for runtime code generation (for example the creation of properties by the use of annotated getters) is not backed by an equally powerful error messaging and caused unnecessary complications without value-adding functionality.

The idea to use annotated methods, as done in ActorFoundry and ActorsGuild, is convenient as it avoids any kind of looping or event-handling by the user. In consequence however, it spreads the messaging-related code throughout the application class, resulting in a tight coupling of application and messaging framework compared to Jetlang, Kilim and Korus. Jetlang's publish-subscribe mechanism is powerful and unique but enforces initial code overhead which eventually pays off considering the extension of the application at a later point. Its 1:1 assignment of fibers to actors limits the scalability of implementations when considering large numbers of interacting entities.

Korus is the light-weight in the comparison. A fixed number of worker threads – per default the number of CPU cores – handle all messaging and are supplied by one or more central schedulers. However, the implementation does not reach the throughput of most other frameworks but its usage is simpler than any other framework. Additionally it provides network support which is not only restricted to platforms of its kind.

3.7 Performance

[9]This implies the support for distributed actors.

[10]Latest release data: 12th of May 2008.

[11]The current version 0.6 is dated 6th February 2009.

[12]Its current version is 1.0 and dated 4th February 2010.

In this section we will expand our previous listing of platform features by a more detailed analysis of the various performance indicators relevant for each of the frameworks discussed. Although comparisons of some of the frameworks have been undertaken (e.g. [22] and [25]), the scenario used here is driven by the intention to measure performance of agent-like interactions rather than measuring performance of sequential or parallelised activities. In order to test this we constructed a scenario resembling massive concurrent interaction of independent entities fully driven by the message passing frameworks:

Numbers, chosen to represent some state, are assigned to agents on a quadratic 2D grid. Every grid cell is 'owned' by an agent. Upon initiation a given ratio of agents randomly request a *swap* of the state with other agents. Along with the swap itself the originally requesting agent becomes idle and the swap target requests the next swap. This is repeated until the first agent reaches a certain threshold for the number of successful swaps. The agents need to ensure that their state is consistent, i.e. transactions need to be fully processed; agents in a transaction refuse requests for a swap. In order to make the use of the frameworks comparable for agent frameworks using a message container for message exchange a (close to) unified message structure is used in all implementations.

Values chosen for the number of rounds are 1000, the number of initially activated agents is 10 percent (Example for grid dimension 100: 10000 agents, 1000 activated initially). The grid dimension represents the independent variable and ranges from 5-500, resulting in 25 to 250000 agents. For the runs Java heap memory usage is restricted to a maximum of 1024MB on an Intel QuadCore PC with 2.66 GHz and 3.25 GB RAM.[13] Measured criteria of the message passing frameworks were performance along with fairness. Performance is measured as sum of all (successful and unsuccessful) swap requests of all swap agents divided by seconds. Each request consists of two messages, thus allows the measure of messages per second.[14]

Fairness is measured as the standard deviation of swaps of all agents. Unfairness is measured as the relative difference between the minimum and maximum number of swaps performed. The results of this benchmark, shown in Figure 2, reveal the largely differing performance results. ActorFoundry and ActorsGuild are the only frameworks advocating safe message passing but also allow in-memory message passing and has thus been included twice clarifying the effects of safe message passing, both having Java serialization as their bottleneck.[15] All other frameworks rely in in-memory passing which allows significant performance boosts. However, given the nature as infrastructure of agent platforms, developers of the latter need to take this into account if they intend to utilize this performance gain. ActorFoundry also outperformed all other frameworks for a small number of agents. However, subsequently the performance declined rapidly with increasing gridsize (and as such increasing number of agents). Key consideration for this is that ActorFoundry was more sensitive to memory adjustments than any other framework, thus garbage collection influenced the results earlier than in other frameworks. ActorsGuild's results were the overall poorest, whether using safe message passing or in-memory; it was not able to handle more than 2500 agents and stopped execution without any feedback. Positive however is the support of multiple message passing modes which not been considered by most others.

Korus took the mid-range position in this benchmark and did not commit to the low-end group nor the leading field. It nearly constantly performed half as fast as Kilim; along with its fairness results this indicates a similar scheduling approach of both frameworks. Retracing the sharp decline in performance for ActorFoundry, the overall fastest framework was Jetlang. Kilim's performance is slightly lower but converged with Jetlang with increasing grid size.

	Kilim	Jetlang	Actor Foundry	Actors Guild	Korus
100	25.1	465	431	425	25.5
225	25.4	446	432	433	24.3
10000	24.7	431	432	-	26.4
40000	25.7	232	232	-	27
160000	25.7	115.8	115.8	-	27
250000	25.7	92.67	96	-	27

Table 2: Fairness of Message Passing Frameworks for selected scenario rounds

In contrast to performance the fairness results[16] clearly indicated two clusters, one with considerably fair frameworks ranging around a standard deviation of 25. Frameworks belonging to this cluster are Kilim and Korus. As both achieved similar fairness results, this value seems to represent the underlying Java random number generator (for the selection of swap agents) rather than framework-specifics. All other frameworks can be considered unfair as in nearly every case at least one agent had not been able to perform even one swap. With increasing rounds the unfairness declined steadily for Jetlang and ActorFoundry. Overall their fairness values were very similar, pointing to a delivery in best-effort manner without central coordination - and as such basically the opposite of Kilim and Korus for the case of fair frameworks.

Given those results, for a practical use in the case of purely reactive agents, Kilim and Korus seem to be the better choice for implementation - in a trade-off of high performance versus the introduction of an additional post-compilation step. However, considering the application in the context of pro-active agents a separate scheduler ensuring execution fairness would allow the consideration of other frameworks - given that messages sent are eventually delivered.

4. CONCLUSIONS

[13]The runs were done using J2SE 1.6 Update 20 on Microsoft Windows XP Professional SP3. The values taken represent the mean of 10 runs for each platform/configuration combination.

[14]The performance calculation does not consider administrative messages such as the initialization and the transmission of results to a central agent. The timing stops once the first agent reached the required number of swaps.

[15]In fact the curves representing safe messaging (serialization) for ActorFoundry and ActorsGuild are overlapping.

[16]For ActorFoundry and ActorsGuild only the fairness values for in-memory message passing are shown as the ones from serialization are not significantly different.

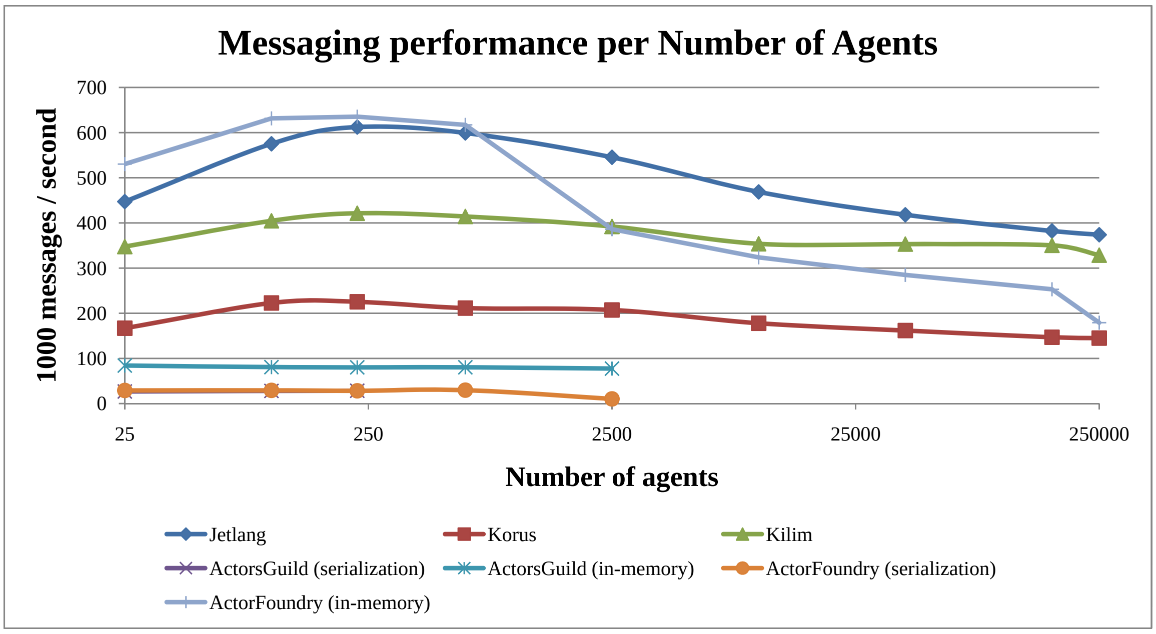

Figure 2: Performance benchmark results Asynchronous Messaging Frameworks

This paper advocates an increasing relevance of micro-agent concepts to converge different fields of agent-based computing by providing more flexible agent platforms. Yet only two platforms of the selection, although with different emphasis, rely on this approach to lower the threshold for the uptake of agent-based software development while allowing the agents to 'grow' with the demands of the application. Making the point for micro-agents to exist alongside with traditional agent concepts, we provide a survey on candidate infrastructure for fast message transport on agent platforms. As a future step alternative threading models respectively concurrency handling mechanisms will be evaluated with regards to their usability in the given context. We hope to encourage further research supporting the notion of micro-agents pioneered by research in the context of OPAL and by others. It should be driven by the spirit to finally convince software engineers to make use of the potential provided by agent-oriented software engineering and remove the stigma of agent-based systems as being impractical, slow and strictly confined to the academic realm.

5. REFERENCES

[1] 3APL Homepage. http://www.cs.uu.nl/3apl/. Accessed on: 25th July 2010.

[2] ActorFoundry. http://osl.cs.uiuc.edu/af/. Accessed on: 25th July 2010.

[3] Actors Guild Framework. http://actorsguildframework.org/. Accessed on: 25th July 2010.

[4] Java Agent Services. http://sourceforge.net/projects/jas/. Accessed on: 25th July 2010.

[5] Jetlang. http://code.google.com/p/jetlang/. Accessed on: 25th July 2010.

[6] Korus. http://code.google.com/p/korus/. Accessed on: 25th July 2010.

[7] Web Services Architecture. http://www.w3.org/TR/ws-arch/. Accessed on: 25th July 2010.

[8] Publicly Available Implementations of FIPA Specifications. http://www.fipa.org/resources/livesystems.html, 2003. Accessed on: 25th July 2010.

[9] JADE - Java Agent DEvelopment Framework. http://jade.tilab.com, October 2009. Accessed on: 25th July 2010.

[10] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[11] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, Englewood Cliffs, 2nd edition edition, 1996.

[12] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade - a java agent development framework. In R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 125–148. Springer, 2005.

[13] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

[14] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. *Third International Conference on Multi-Agent Systems (ICMAS '98), IEEE Computer Society*, pages 128–135, 1998.

[15] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[16] FIPA. FIPA Agent Management Specification. http://www.fipa.org/specs/fipa00023/SC00023K.html, 2004. Accessed on: 25th July 2010.

[17] O. M. Group. The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group, 1998.

[18] O. Gutknecht, J. Ferber, and F. Michel. The MadKit Agent Platform Architecture. Technical report, Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, Universite Montpellier II, 2000.

[19] M. Henning. The Rise and Fall of CORBA. *Communications of the ACM*, 51(8):53–57, 2008.

[20] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[21] N. R. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. *Artificial Intelligence*, 117:277–296, 2000.

[22] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *7th International Conference on the Principles and Practice of Programming in Java*, 2009.

[23] T. Malsch, C. Schlieder, P. Kiefer, M. LÃijbcke, R. Perschke, M. Schmitt, and K. Stein. Communication between process and structure: Modelling and simulating message reference networks with COM/TE. *Journal of Artificial Societies and Social Simulation*, 10(1), 2007.

[24] M. Nowostawski, M. Purvis, and S. Cranefield. KEA - Multi-Level Agent Architecture. In *Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS 2001)*, pages 355–362. Department of Computer Science, University of Mining and Metallurgy, Krakow, Poland, 2001.

[25] S. Pal. More Java Actor Frameworks Compared. http://sujitpal.blogspot.com/2009/01/more-java-actor-frameworks-compared.html, 2nd January 2009. Accessed on: 25th July 2010.

[26] M. P. Singh. Agent Communication Languages: Rethinking the Principles. *Computer*, 31(12):40–47, 1998.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *European Conference on Object Oriented Programming ECOOP 2008, Cyprus*, 2008.

[28] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), 2005.

[29] J. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., Mountain View, CA, USA, 1994.

[30] J. E. White. RFC 707: High-level framework for network-based resource sharing, Dec. 1975. Status: UNKNOWN.