

Technical Report: TUB-DAI 07/09-01

MicroJIAC 2.0 - The Agent Framework for Constrained Devices and Beyond

Marcel Patzlaff Erdene-Ochir Tuguldur

July 2, 2009



CC ACT
Agent Core Technologies

Abstract

MicroJIAC 2.0 is a matured version of a JAVA-based agent framework born and raised in a diploma thesis. Its development followed a bottom up approach starting with resource constraint devices. Consequently MicroJIAC is also usable on more capable devices. Experiences from projects using the framework lead to modifications of the basic concepts. All derived requirements could be met while retaining the broad field of application.

1 Introduction

MicroJIAC was specified and developed in 2006 and is the result of a diploma thesis [Pat07]. The objective was to create an agent framework usable on devices with minimal JAVA support (CLDC 1.1 [Sun03]). Unlike other frameworks at this time, we followed a bottom-up approach and evaluated the constraints of this weakest JAVA configuration and class of device, and designed MicroJIAC to fully exploit all given features, instead of downsizing an existing desktop framework. Several issues were solved by delegating them into the build phase of a MicroJIAC project:

- the size of the application's JAVA archive (JAR) is reduced by using the ProGuard [Laf] tool. Unused classes are removed from the archive and class names are shortened.
- parsing and processing of the application's configuration is done at compile time. Thus an XML parser is not needed at runtime.
- CLDC has no reflection mechanism. However, to support runtime modifications of the application a reflector class is composed and included into the application's JAR.

Those and other tricks increased the commonalities of all JAVA editions. It was feasible to broaden the field of application and to build a framework supporting most of the JAVA platforms that are at least compatible with CLDC 1.1. MicroJIAC agents, developed for CLDC-based JAVA editions, are also runnable on J2SE and J2EE without modifications.

Since the first versions, MicroJIAC underwent several serious modifications and extensions to meet further requirements:

- real-time support [BBD⁺00]
- adaptable nodes and agents
- "hot deployment" and migration
- increased usability

The concepts that fulfill those requirements are introduced and explained in the following chapter.

2 Concepts

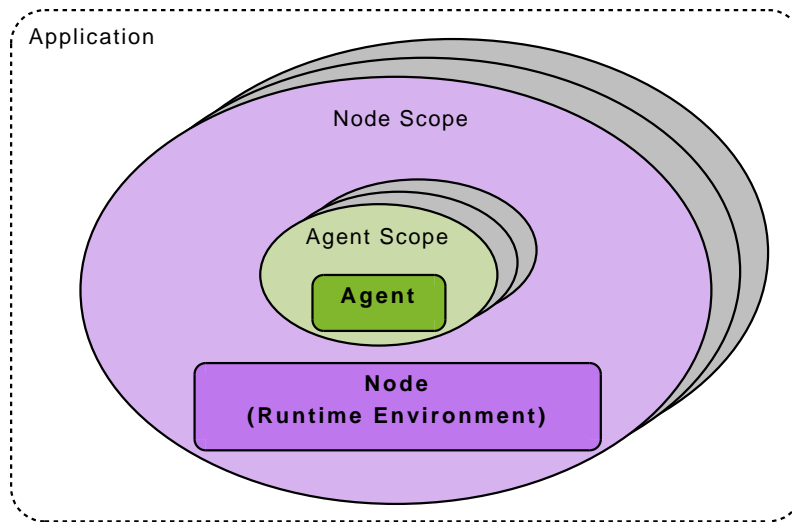


Figure 2.1: Basic Concepts in MicroJIAC

In MicroJIAC an application always consists of one or more nodes hosting one or more agents. *Applications* are virtual constructs to group nodes and agents together and to document their associations. The other entities of MicroJIAC, depicted in Figure 2.1, are explained in the following sections.

2.1 Node

A *node*, depicted in Figure 2.2, is the runtime environment for agents and provides initialisation and startup routines. Each node is associated with a single JAVA virtual machine so both terms are synonymous in the MicroJIAC context. Basically the node abstracts from particular JAVA libraries located on the different devices. For example, essential and platform-dependent resources like network connections can be obtained through the node. The mechanism to do that is borrowed from the “Generic Connection Framework” [Ort03] which associates specific connection types with URIs (Uniform Resource Identifiers). Node customisations through specific components may extend the pool of resources obtainable by agents.

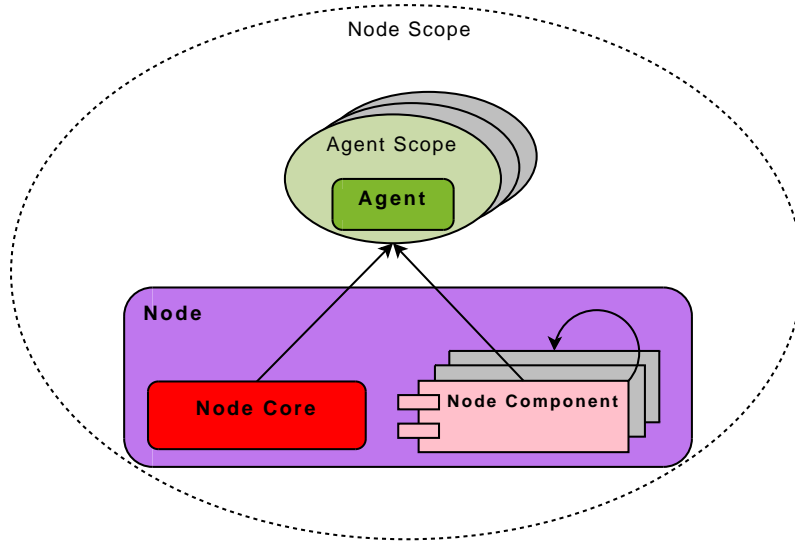


Figure 2.2: Structure of a Node. The black arrows depict the direction of handles to node components or agent scopes.

The node is responsible for creating a scope for each agent and to configure and execute it according to the provided agent configurations (see Section 2.7). This ensures that agents cannot obtain direct references to node-internal components and functions but only through handle interfaces. Those handle interfaces are provided by specialised node implementations or node components. We present more details to handles and scopes in Sections 2.3 and 2.4.

2.1.1 Node Component

Each node is extensible through *node components*. This extension mechanism is a means to encapsulate functionalities of the host system and to provide them to the node itself and to all agents running on it. The node developer can chose freely whether to provide system interfaces or to specify abstractions. A communication system for example is an abstract node extension which realises an agent communication channel. Access to it is granted through a defined handle interface for every agent on the node, though the agent developer does not have to bother with details of the communication technology in use. Other nodes may provide the same interfaces for the agent communication channel but with different implementations beneath.

A system-dependent node extension could be, for example, a connection to a database available only on the specific host. Such dependencies might be reflected in the handle interfaces and might also influence the runnability of the agents.

2.2 Agent

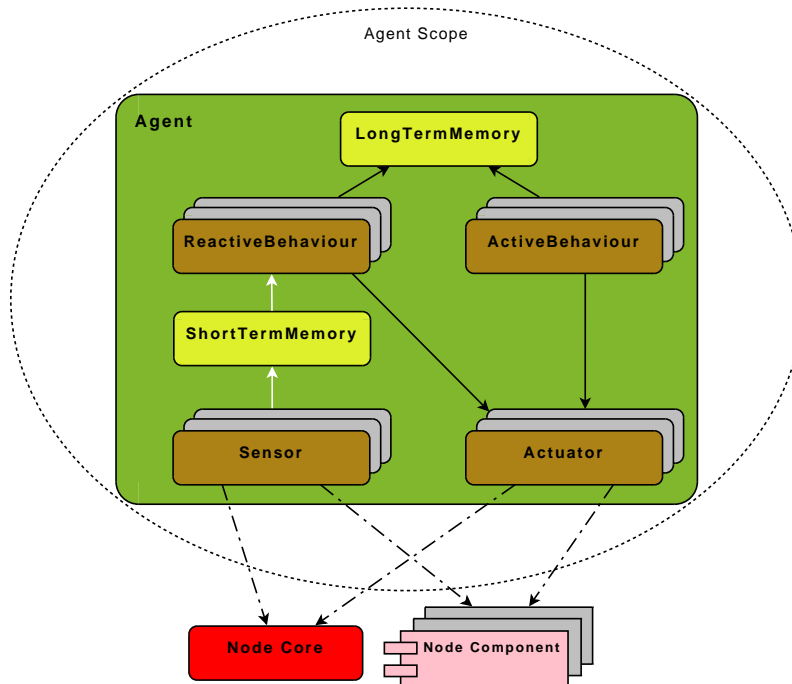


Figure 2.3: Composition of an Agent. Black arrows depict the direction of handles to other agent or node components. White arrows depict special cases of agent-internal accesses.

Agents (see Figure 2.3) are the entities in MicroJIAC doing more or less useful things. They are composed of so-called agent elements which MicroJIAC distinguishes four kinds of: **sensors**, **actuators**, **reactive behaviours** and **active behaviours**. Furthermore there is a simple management layer within every agent. This layer is responsible for the correct initialisation and state transitions of the installed elements and also for dynamic installation and removal of functionalities.

Usually the agent developers do not need to implement their own agent classes or to extend the concept. In case, where there is no other solution, the developer should keep in mind that an extension or modification of the agent class might constrain the portability of the actual agent functionalities.

2.2.1 Knowledge Representation

Agents can only process information if there is a vocabulary to describe this information. The vocabulary is collected in so-called *ontologies* and reflects the agent's environment as well as agent-internal events. MicroJIAC offers an abstract **Fact** type

whose specialisations encapsulate the world which can be experienced by the agent. At the moment, the developer is responsible to describe the domain with an appropriate data model and to build JAVA classes for it. Due to the restrictions of CLDC, a dedicated ontology description language like OWL [W3C04] is too expensive, with respect to memory consumption and computation costs. Instead, ontologies are directly implemented as JAVA classes with several attributes that are accessible through the usual getter and setter methods.

There are two different ways to “generate” knowledge: A sensor gets input through readings in its sampling area and transforms it to shape it into new knowledge for the agent’s intelligence. On the other hand, behaviour-elements obtain new insights due to internal activities and provide them for further procession and storage. Those internal activities can be background computations or reactions.

The knowledge representation using JAVA classes enables the interplay between agent elements, meaning the agent-internal flow of information, and the information exchange between multiple agents.

2.2.2 Memory

In MicroJIAC we differentiate information storage in short-term and long-term. The *short-term memory* is only accessible for the sensors and forms an “interim storage” for new information. This information is provided to the reactive behaviours (see Section 2.2.4). If it is not used in those behaviours, the information will be “forgotten” immediately. This approach limits the amount of data in the virtual machine’s memory and reduces the risk of memory leaks.

Contrary to that, all data in the *long-term memory* is stored until it gets removed explicitly. Hereby, the correct handling is up to the agent developer, because he knows best, which information is essential in the long run. All elements of the agent have (read and write) access to the long-term memory.

2.2.3 Sensor and Actuator

According to a standard work regarding “artificial intelligence” [RN03], MicroJIAC’s agent model also offers elements that allow the manipulation and reading of the agent’s environment:

Sensors actively sample the environment or get notified by it and provide the collected data to the short-term memory of the agent. Their counterparts, *actuators*, push data into the environment and/or manipulate it. Structure and content of the data generated by sensors is explicitly specified in the knowledge representation.

Information, exchanged with the agent’s environment, is translated in sensors and actuators. These data transformations uncouple planning and reaction functionalities from data retrieval and thus increase device independence.

Each actuator defines a handle interface through which other agent elements can access it. The agent developer should ensure that only platform independent functionalities are reflected by the interface and that internal agent logic remains hidden.

2.2.4 Behaviours

MicroJIAC defines two different types of behaviour. The *reactive behaviour* works on data provided by the short-term memory. The rather simple filtering mechanism ensures that reactive behaviours are only triggered if data of the specified types are available. Due to the restrictions of CLDC, there is (currently) no possibility to realise filtering with type information **and** attribute values.

Reactive behaviours are versatile elements. For example they can be used as connectors between sensor and actuator implementing some kind of “reflex logic”. Furthermore they can manipulate the content of the long-term memory influencing the active behaviours of the agent.

Execution of reactive behaviours is decoupled from the appearance of new data in the short-term memory, hence asynchronous to the sensor’s read operations. This approach is a consequence of MicroJIAC’s real-time capabilities and is presented in more detail in Section 2.6.

The second type of behaviours is the *active behaviour*. It works periodically on data of the long-term memory and is able to manipulate the agent’s environment through handles to the actuators. Active behaviours are used to realise higher-order planning logics and functionalities. Both kinds of behaviour, reactive and active, form the “intelligence” of each agent and should always be realised without introducing any device specific dependencies. Only this design ensures that behaviours are exchangeable and portable to other agents and devices.

2.3 Handle

Handles are links to other components. In general, they are not just direct references but views onto the component’s functionalities. In MicroJIAC there are three sources for handles:

1. The node core, where all system dependent functionalities are encapsulated, provides class loaders and resource-aquisition handles.
2. Each node component may provide node-internal handles and/or agent handles. An example is the aforementioned communication system.
3. Each actuator in the agent has to define and provide a handle.

Handle interfaces (and sometimes also classes) just have to extend a specific marker interface and are made available automatically. Developers should keep handle definitions as abstract as possible to increase decoupling of agent elements and node components.

2.4 Scope

Scopes encapsulate agents and nodes and prevent direct references onto these instances. This is an essential requirement of the real-time specification for JAVA

(RTSJ) [BBD⁺00] as it allows the usage of different memory areas. A scope provides access to its assigned memory area and to the container reference (which is either the node or the agent).

Each thread, other than the initial bootstrap or system thread, is associated with a scope. Node threads belong to the node scope and agent threads to the agent scope respectively. We used a specialisation of the `java.lang.Thread` class and made it *scope aware*. During execution of node or agents, threads must only be created through the provided factory. Otherwise the logic, that runs in the new thread, cannot access scope related attributes or references.

The scope concept is present in each MicroJIAC-edition but only the real-time one fully implements it.

2.5 Lifecycle

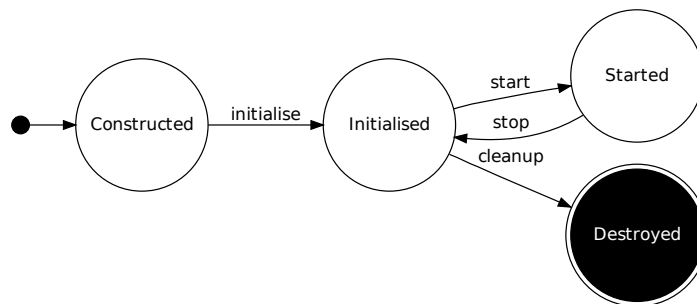


Figure 2.4: Lifecycle in MicroJIAC

MicroJIAC defines a simple *lifecycle* for nodes, agents and agent elements. It contains four states: **constructed**, **initialised**, **started** and **destroyed** and is depicted in Figure 2.4. By default, nodes and agents are governed by a lifecycle manager. There is a specific **LifecycleAware** interface which defines hook methods for the state transitions. The appropriate methods are called whenever the manager issues state transitions. Agent elements can also be included into the lifecycle management by extending that same interface.

2.6 Scheduling

Scheduling is an integral part of every real-time enabled framework and thus also of MicroJIAC. Subject of scheduling are runnables which are available in periodic and sporadic variants. The following parameters are interesting for the scheduler:

- deadline - maximum duration of the runnable
- priority - ordering information
- costs - the estimated duration of the runnable

Periodic runnables provide two additional parameters: the start time — time of the first execution — and the period — the time distance between two subsequent executions.

Sporadic runnables instead extend the parameter set with the minimum interarrival time — the minimum time distance between two subsequent executions. Details to the scheduling go beyond the scope of this technical report. We only emphasise that the real-time edition of MicroJIAC handles time constraints much more restrictive than it is the case in the normal editions. The latter do not allow concrete predictions of the runtime behaviour.

The aforementioned behaviour elements are specialised runnables. Active behaviours always realise periodic runnables whereas reactive behaviours are always sporadic runnables.

2.7 Configuration

An application is always as much as its description or configuration. MicroJIAC uses an XML-based configuration language and provides an abstract description model. This “Abstract Agent Meta Model” distinguishes between application, node, agent, agent element and ordinary object. This separation eases the composition of agent-oriented applications.

The deployment or execution of applications depends on the JAVA platform in use. MicroJIAC for MIDlet-based architectures (e.g. mobile phones, SUNSpots, etc) needs to preprocess the configuration to generate JAVA descriptors from it. This avoids the overhead of parsing and processing XML on the device. The JAVA descriptors are then used to configurate and launch the node with all specified agents on it. Instead, the JAVA 2 SE version of MicroJIAC parses the XML configuration file at runtime.

3 Conclusion

MicroJIAC is used in several research and development projects. Those projects are valuable sources to evaluate and mature the framework. With MicroJIAC 2.0, we now have a stable framework that enables agent-oriented development and eases the implementation of device independent functionalities. The support of many different device classes, bounded only to a JAVA virtual machine, gives us a broad field of application.

Although the framework APIs are stable, there is still much work to do. Enabling concepts in MicroJIAC, like real-time and adaptability support, now need to be exploited. Currently we are working on the MicroJIAC edition which fully supports and uses the RTSJ. Furthermore we use MicroJIAC to build an adaptive framework with increased fault tolerance. Both works will be outlined in more detail in upcoming publications.

Bibliography

- [BBD⁺00] Gregory Bollela, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [Laf] Eric Lafortune. ProGuard. available at <http://proguard.sourceforge.net/>.
- [Ort03] C. Enrique Ortiz. The generic connection framework, aug 2003. available at <http://developers.sun.com/mobility/midp/articles/genericframework/>.
- [Pat07] Marcel Patzlaff. Development of a Scalable Agent Architecture for Constrained Devices. Master's thesis, Technische Universität Berlin, April 2007.
- [RN03] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd Edition edition, 2003.
- [Sun03] Sun Microsystems, Inc., Sun Microsystems, Inc. - 4150 Network Circle - Santa Clara, California 95054 - U.S.A 650-960-1300. *Connected Limited Device Configuration*, specification version 1.1 edition, March 2003. available at <http://jcp.org/aboutJava/communityprocess/final/jsr139/>.
- [W3C04] W3C. *OWL Web Ontology Language*. The World Wide Web Consortium, Februar 2004.