Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2011, Article ID 616838, 13 pages
doi:10.1155/2011/616838

*Research Article*

# Peer-to-Peer Jini for Truly Service-Oriented WSNs

**António Pereira,[1,2] Nuno Costa,[1] and Carlos Serôdio[3]**

[1] *School of Technology and Management, Computer Science and Communications Research Centre, Polytechnic Institute of Leiria, 2411-901 Leiria, Portugal*
[2] *INOV INESC INOVAÇÃO, Instituto de Novas Tecnologias, Delegação de Leiria, 2411-901 Leiria, Portugal*
[3] *CITAB - Centro de Investigação e de Tecnologias Agro-Ambientais e Biológicas, Universidade de Trás-os-Montes e Alto Douro, Quinta de Prados, Apartado 1013, 5001-801 Vila Real, Portugal*

Correspondence should be addressed to Nuno Costa, nuno.costa@estg.ipleiria.pt

Received 16 September 2010; Revised 13 January 2011; Accepted 20 April 2011



In the past, wireless sensor networks emerged and so did some state-of-the-art applications that ran above them. Most of those applications followed the centralized model and were mainly focused on environmental data gathering, where sensor nodes sensed data from the environment to be sent to an external data server for later processing. Further research took wireless sensor networks to new application areas and, today, this technology gained the ubiquitous status. However, the use of wireless sensor networks is still reserved to experts, mainly due to the huge gap between user applications and the network. This fact has led us to successfully develop a new software stack, capable of running in any sensor node even in the most resource-constrained ones. The new software stack offers a truly service-oriented approach to design and implement applications and follows a widely accepted programming language and programming paradigm.

## 1. Introduction

The advances in wireless communications and micro-electromechanical systems (MEMSs) together with low-cost CMOS technologies made the development of affordable small-sized battery-powered sensor nodes possible. These wireless nodes, equipped with sensors, actuators, processor, memory, and wireless communication interface, allowed the deployment of wireless sensor networks (WSNs) to support applications such as large-scale environment monitoring, medical care, target tracking, intrusion detection, and surveillance, among others. According to [1], environmental monitoring is the prime application field of the WSNs.

Sensor nodes are extremely resourceful constrained devices powered by batteries. Due to this, many of the researchers on WSNs focused their efforts on minimizing the energy consumption of the whole network. Techniques such as data compression, data fusion, and other optimized data delivery schemes, Time Division Multiple Access- (TDMA-) oriented communications, and enhanced MAC protocols, are commonly used as they already proved [2–7] to be advantageous in energy savings.

Recent research clearly made this kind of networks gain the ubiquity status. The evidence of this is that WSNs have been referred as the underlying infrastructure to support the so-called ubiquitous computing [8].

Ubiquitous computing (UC) is the new computing era envisioned by Weiser and brown [8]. UC is characterized by the connection of things in the world (comprising computation and communication) at many scales, including the microscopic one [8]. Computing will be embedded in walls, chairs, clothing, light switches, and cars—in everything [8]. The ubiquitous computing paradigm envisions embedded computation, operating in heterogeneous spaces and objects of the physical world, where devices and services are easily "plugged-and-played" and cooperate among themselves, spontaneously, to bring convenience to our lives.

Another emergent field of applications for WSNs is the emergency rescue where the sink-less "topology" is the more applicable and where rescue people carry hand-held devices to query network directly. The networks used in this scenario must be easy to setup and the provided services must become available immediately or as soon as possible.

Researchers believe that the Service-Oriented Architecture (SOA) can bring enormous benefits to the WSNs because it can turn these networks into open, ubiquitous, interoperable, and multipurpose infrastructures [9]. The

programming task becomes easier because sensor node capabilities are abstracted and defined as services and applications are written based on service requests issued to the network. However, the resource-constrained nature of sensor nodes has negatively influenced the proposed SOA solutions, since the majority of them rely on in-the-middle resource rich devices to make the bridge between the outside world and the WSNs. These SOA solutions are not suitable for the latest application scenarios described above (e.g., emergency rescue). The works of [10–15] are examples of such SOA solutions.

Another limitation seen in these unsuitable solutions is that sensor nodes are service providers only. This means that an external device can interact with sensor nodes, using the service-oriented approach, but nodes use another paradigm to interact among themselves, if needed.

The work presented in this article, called Peer-to-Peer (P2P) Jini, overcomes these limitations. P2P Jini refers to a software stack which includes Hardware Abstraction Layer (HAL), Operating System (OS) Layer, communication layer, and middleware layer that follows the Jini [16] SOA specification. While the state-of-the-art TinyOS operating system adopts the event-based programming model and the NesC programming language and introduces a high learning curve for most of the programmers, P2P Jini adopts the wide accepted Java programming language, the Java programming paradigm, and the Jini service-oriented architecture, also belonging to the Java World.

P2P Jini was presented for the first time in [17] as simple proof of concept, where the main aims were to highlighting its functional model and its strengths. A minimal set of simple test cases were also evaluated. This article aims to describe and evaluate further P2P solution and present it as a valid solution to be used in sink-less wireless sensor networks applications, like emergency rescue.

The rest of the article is structured as follows: Section 2 presents the limitations of the current service-oriented architectures; Section 3 gives a high-level overview of the Peer-to-Peer Jini and establishes its main requirements; Section 4 describes the minimum target hardware platform characteristics; Section 5 details the Peer-to-Peer solution and its relevant characteristics; Section 6 presents the solution evaluation; Section 7 describes the relevant related work and finally Section 8 summarizes the contributions of the work presented in this article.

## 2. Limitations of Current SOA Solutions

Current SOA solutions proposed to address resource constrained sensor nodes have the limitations summarized as follow.

(1) High level learning curve—for instance, service-oriented approaches based on TinyOS operating system, such as TinySOA [18], always impose a high learning curve.

(2) Need for resource rich (external) device(s): usually, SOA solutions proposed to address sensor networks always rely on the network gateway device to implement the service-oriented layer. This approach has four main problems: (i) it is a centralized architecture which suffers from the "single point of failure" and demands more effort on network communication, (ii) these kinds of solutions are not suitable for sink-less (emergency) sensor networks because it does not make sense to set up a resource rich device in order to interact with the network (time to set up), (iii) it is not compliant with mobility because if a node arrives or departs, it is necessary to inform gateway about that in order to have a refreshed view of the whole network, and (iv) it usually requires two different programming contexts from the programmer; for example, the programming of nodes is usually accomplished using a low-level language, and the user applications are programmed using a well-known SOA paradigm like Web Services. The gateway makes the bridge between both worlds.

(3) Service providers only: the existing wireless sensor networks SOA approaches always transform sensor nodes into service providers. This, of course, guarantees network queries and actions. However, sensor nodes just react to "outside" orders and this is not suitable for ubiquitous spaces when the devices may interact with each other spontaneously.

(4) Poor-service-oriented approach: unlike P2P Jini, which has a pure and clear service-oriented programming approach, other proposals usually have a poor service-oriented programming model and the programmer does not often have any perception that it is dealing with services. Usually, these approaches need many more lines of source code and service discovery and service invocation operations are not explicit.

(5) Heavy communication: most of the existing wireless sensor networks SOA proposals adopted the Web Services approach due to its popularity, openness, and interoperability. However, Web Services demand much effort to decode messages and consume more bandwidth than binary protocols.

P2P Jini was designed and implemented to overcome the main drawbacks (described above) of the existing SOA solutions for wireless sensor networks.

## 3. The P2P Jini

P2P Jini comprises a Java-based software stack that can run virtually on any device, ranging from resource-constrained sensor nodes to desktops and servers. P2P Jini stack is composed of a Hardware Abstraction Layer, a Java Virtual Machine (which behaves as an operating system), a communication stack based on TCP/IP, and the P2P Jini layer, which is a Jini-like service-oriented middleware.

*3.1. Java Virtual Machine Layer.* P2P Jini is based on Jini [16]; hence it naturally inherited the need for an underlying

Java Virtual Machine. The Java Virtual Machine layer, which is represented by the JVM128 Java Virtual Machine, was developed using C and assembly language and borrows code from other open source Java Virtual Machines, such as leJOS [19], TinyVM [20], and Waba [21]. JVM128 is composed of three main modules. The first module is the Hardware Abstraction Layer, which results from the lack of an explicit operating system. This module aims to make the bridge between hardware and the operating system more precisely than between the hardware and the JVM128. The HAL layer is especially important because it allows the high-level Java language to communicate with the hardware by abstracting it with well-defined functions.

The second module is the TCP/IP stack. TCP/IP stack is implemented using C language and is embedded inside the JVM128 in order to save hardware resources. The API to address it is offered in the form of Java native methods. Of course, a Java implementation of the TCP/IP would be more advantageous in terms of porting operations, but it would consume much more resources.

The third module is the Java Virtual Machine itself. JVM128 follows the interpretative execution model; it is stack-based and has a very low-memory footprint. JVM128 byte code is compliant with the byte code defined by SUN Microsystems for its Java platform. This means that the Java programs to be interpreted by JVM128 can be developed and compiled using standard Java development tools. However, due to its target use, JVM128 only supports a subset of the Java language. In terms of byte code, JVM128 only implements the standard ones (from 0 till 201), with the exceptions for long and double data types, exceptions, and threads. JVM128 occupies less than 40 KB of program memory (ROM) and less 350 bytes of data memory (RAM).

*3.2. Peer-to-Peer Jini Layer.* The Peer-to-Peer Jini layer runs above the Java Virtual Machine layer and implements a different approach to support automatic service discovery and service invocation, when comparing it with standard Jini. P2P Jini is so lightweight that even a resource-constrained device like MICA2 can act as a service provider, as a service requester or both. To the best of our knowledge, other SOA solutions only support the service provider role, almost always relying on external resource rich devices.

To be viable, P2P Jini must respect the following requirements.

(1) Hardware independency: this requirement is accomplished by the use of the underlying Java Virtual Machine. This way, any Peer-to-Peer Jini application can run virtually on any hardware platform, ranging from sensor nodes to high-end servers.

(2) Low-level learning curve: P2P Jini must adopt widely used programming language and programming paradigms. For instance, the state-of-the-art TinyOS operating system has a high-level learning curve because it is event driven and adopted the NesC (a dialect of C) as programming language.

(3) Lightweight solution: the whole software stack needed to run Peer-to-Peer Jini services/applications must consume few resources in order for the stack to address even the most resource-constrained hardware.

(4) Ubiquitous communication protocol: P2P Jini must adopt the most used communication protocol(s), for example, TCP/IP, in order to maximize the compliance with other computing devices with communication capabilities.

(5) Fully service driven: P2P Jini must allow a device to behave as a service provider, as a service consumer, or both.

(6) Ad hoc compliant: P2P Jini must be compliant with the challenges posed by the ad hoc networking. This means that P2P Jini must be prepared to gracefully handle node arrivals and departures without spending any extra resources. This requirement was met by the adoption of a distributed architecture instead of the centralized architecture used by standard Jini. While Jini relies on a centralized Lookup Service and centralized Code Base Provider, P2P Jini spread these core services across the whole network. This eliminates the "single point of failure" problem and makes the whole network more robust. For instance, while Jini uses leases to control node/service failures, P2P Jini can instantaneously detect when a new service arrives or when an existing service becomes unavailable, without spending any extra resources. Jini leases demand renewal operations of leases (wasting communication messages) and new service availability and service unavailability suffer from delays.

## 4. Hardware Platform

The MICA2 sensor node, from Crossbow Technology Inc [22], is the reference hardware platform. MICA2 uses an 8-bit ATMEL ATMega 128 L microcontroller with 4 KB SRAM (RAM), 128 KB FLASH (ROM), 4 KB of EEPROM, a CC1000 radio chip, and several ready-to-use on-board sensors. Because the development will deal with hardware specificities, a development board, compliant with the target hardware, was acquired in order to ease development, testing, and debug processes (STK300 kit from Kanda [23]).

The STK300 Kit includes all the needed pieces to develop designs for AVR ATMega128x devices. More concretely, it includes support for Liquid Crystal Display (LCD) interface, 3.3/5 V operation, real-time clock support, external flash RAM support, switches and LEDs, 24 C EEPROM socket, In System Programmer (ISP) and JTAG interfaces, and two Recommended Standard 232 (RS232) ports. An LCD was used for debugging purposes only. To program the microcontroller the WINAVR AVR C compiler with the assistance of the Avr Studio-Integrated Development Environment (IDE) can be used. A big advantage of using the Avr Studio IDE is its integrated emulator, which avoids having to download code to the chip every time one needs to test

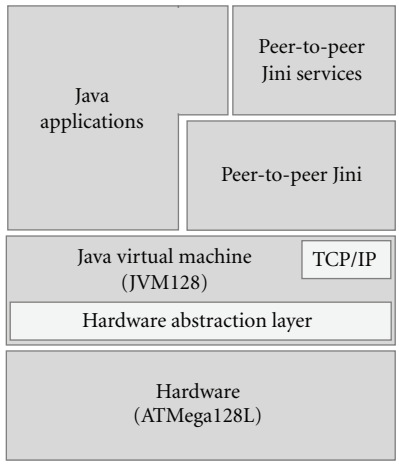


Figure 1: The whole P2P software stack.

it. To provide communications, a Transceiver Development Node was acquired (nRF2401A with a Transceiver nRF2401A Module Trace Antenna) from Sparkfun [24]. The Transceiver Development Node behaves like a bridge to the transceiver module by supplying a serial port to drive the transceiver in an easy manner. This development board embeds a PIC microcontroller that interfaces with the transceiver module. Once programmed, the user has just to send (or read) data to the serial port in order to send (or receive) data through the air. However, due to the lack of resources of the Transceiver Development Node, it was removed and the Transceiver module was driven directly by the ATMega microcontroller.

## 5. The Details of the Peer-to-Peer Jini Solution

Although P2P Jini refers to the whole software stack, this section focuses only on service-oriented approach implemented by the Peer-to-Peer Jini layer (see Figure 1).

*5.1. Standard Jini Overview.* Jini is a Java-based distributed computing framework proposed by Sun Microsystems supported and maintained in the Apache River project. Jini provides a programming model (interfaces) and a runtime infrastructure (services) to achieve the concept of a spontaneous network where a Jini-enabled device is plugged in the network and works possibly without human intervention. Since Jini relies on Java, virtually any device (from a toaster to a television set) can offer and require Jini services over the Jini network, also known as Jini federation.

The Jini architecture is comprised of three main elements: the Lookup Service, the Service Provider, and the Service Requester. The Service Provider is the entity that offers the service. The service is offered by making it available at the Lookup Service(s). To achieve that, Service Providers must discover one or more Lookup Services and upload the service-serialized form, which usually is a Service Proxy. Service Requesters, in turn, discover one or more Lookup Services and search for services based on Service ID, Service Interface, or Specific Attributes. If there is a true match, then the Lookup Service sends the Service Proxy (uploaded by the Service Provider) to the Service Requester, where it is then used to invoke the Service maintained by the Service Provider. Typically, interaction between the Service Proxy and the Service is accomplished by using Java Remote Method Invocation (RMI). However, it is not mandatory and it is completely transparent for Service Requesters. The Lookup Service (Service Directory) is the central piece of the Jini architecture. It is the Lookup Service that stores and keeps track of all Jini Services of the Jini federation. For reliability and scalability reasons, there could be more than one Lookup Service per Jini federation.

*5.2. Peer-to-Peer Jini.* The standard Jini solution was developed based on the characteristics of networks with stable infrastructure, while P2P Jini targets ad hoc networks. Since our team has been using Jini for a long time, and due to the fact that we consider it good software, it was decided to adapt it to the infrastructure less networks, while keeping its semantics.

The existing Jini implementations rely on Java RMI for communications between proxies and services. However, the research reported in [25] found that RMI is the Jini bottleneck when considering resource-constrained devices. To avoid the use of RMI it is not enough to replace it with other communication mechanisms because RMI by itself also depends on another important and "heavy" Java mechanism, called Java Serialization. It is Java Serialization that mainly distinguishes RMI and Jini from its competitors. Hence, the RMI must be replaced by another communication model, more suitable for resource constrained devices, but Java Serialization must be preserved at any cost.

*5.2.1. RMI Replacement.* At this point, it is important to note that when a Jini requester is looking or invoking a Jini service, the communication mechanism is completely transparent, due to the use of proxies on the requester side. But proxy downloading requires code and data mobility (some authors [13] decided to remove this mobility facility, paying the price of loosing the Jini requester side, since their solutions only apply to Jini providers). For instance, the RMI can be replaced by any lightweight remote procedure call, but, again, mobility must be kept. Considering the resources of the target platform, even the smaller ready-to-use RPC solution (e.g., XML-RPC [26]) does not fit in. In our opinion, for this class of hardware, the solution that is left is a pure binary message-oriented protocol for remote method invocation.

Figure 2 depicts the basic operations of the proposed binary message-oriented method invocation protocol. Firstly, client (or service requester) by using the service proxy connects to the server service dispatch port. At this stage, the server accepts the connection and waits for the protocol version the client intends to use. If it is a valid version, the server returns OK; otherwise the server returns ERROR and closes the connection. If the protocol version was accepted, then the client sends the method ID and related arguments. For simplicity purposes, both method ID and the arguments

```
public class LUSandJiniServer
{
  static short m_RemPort=0;
  static int m_proxyclassLen=0;
  static byte m_btState=0;
  static int proxyFileIndex=0;

  public static void main()
  {
    int[] listenPorts = new int[2];
    listenPorts[0]=9930; //For broadcasts
    listenPorts[1]=4030; //For service invocation
    sockets.ServerSocket(2,listenPorts);
  }

  //UDP Socket Handler
  public void onRecvFrom(short target_sock_port,byte orig_ip1,byte orig_ip2,
    byte orig_ip3,byte orig_ip4,byte[] buf)
  {
    if(target_sock_port == 9930)
    {
      int i=0;
      byte len=buf[0]; //Msg len
      byte ver=buf[1]; //Jini version
      m_RemPort=0;
      m_RemPort |= buf[3] & 0xFF; //Decode remote port int value
      m_RemPort <<= 8;
      m_RemPort |= buf[2] & 0xFF;
      //SERVICE NAME
      byte strNameLen = (byte) (len - (byte)3);
      char[] strName=new char[10];
      for(i=0;i<strNameLen;i++) {
      strName[i]=(char)buf[4+i];
      }
      //Test here the service name to see if we get what is looked for
      //and fill m_proxyclassLen variable (...)
      buf[0]=(byte)0; //Clear in/out buffer
      //Connect to caller to then send service proxy
      sockets.Socket(orig_ip1,orig_ip2,orig_ip3,orig_ip4,m_RemPort);
    }
  }

  //TCP Socket Handler
  public void onSockEvent(short local_port, short remote_port,short sock_evt,byte[] buf)
  {
    int i=0;
    byte bt;
    if(sock_evt == 0)
    {//Connected
    if(remote_port==m_RemPort) {//Port to send service is connected
      {
        buf[0]=(byte)0; //Do not reply anything
      }
  }
  if(sock_evt == 1) //New Data
  {
      if(remote_port==m_RemPort)
      {
      }
      if(local_port==4030) //Service request
      {
        bt=buf[1]; //Test Jini version here
```

Algorithm 1: Continued.

```
            bt=buf[2]; //Method number
            buf[0]=(byte)2; //Reply msg has length 2
            if(bt==(byte)1)
            {
              buf[1]=(byte)0; //Return OK
              buf[2]=(byte)LM35.GetTemp();
            }
            else
            {
            buf[1]=(byte)1; //Return Error
            }
            return;
          }
          buf[0]=(byte)0; //Do not reply anything
        }
        if(sock_evt == 2) //Poll
        {
          if(remote_port==m_RemPort) //Port for sending service proxy
          {
            if(m_btState==0) //Send len
            {
              //Send proxy class len
              buf[0]=(byte)4;
              buf[1]=(byte) (m_proxyclassLen>>24 &0xFF);
              buf[2]=(byte) (m_proxyclassLen>>16 &0xFF);
              buf[3]=(byte) (m_proxyclassLen>>8 &0xFF);
              buf[4]=(byte) (m_proxyclassLen &0xFF);
              m_btState=1;
              return;
            }
            else if(m_btState==1) //Send file
            {
              buf[0]=(byte)25; //Max playload
              for(i=0;i<25;i++)
              {
                buf[i+1]=(byte)out.GetFilePos(proxyFileIndex);
                proxyFileIndex++;
                //Store the classbase address here
                if(proxyFileIndex > m_proxyclassLen)
                {
                  proxyFileIndex=0;
                  m_btState = 0;
                  return;
                }
              }
              return;
            }
            buf[0]=(byte)0;
          }
        }
        if(sock_evt == 3) //Resend
        {
          buf[0]=(byte)0; //Do not reply anything
        }
        if(sock_evt == 3) //Closed
        {
            buf[0]=(byte)0; //Do not reply anything
        }
      }
    }
```

Algorithm 1: Complete Application for providing P2P Jini Services (JVM128).

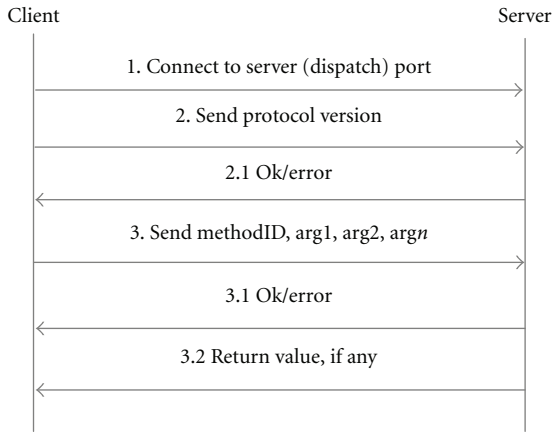


Figure 2: Lightweight remote method invocation protocol.

are in character string format. The server reads the method id and arguments and, if the method id is valid, it will invoke it and return an error code to the client. If the error code is "OK", then the method return value will be returned.

*5.2.2. Serialization Replacement.* The object mobility used in RMI (and Jini), as the name indicates, considers both object data and object code mobility. This means that when an object is moved across the network (a service proxy, e.g.), it carries both the object code and the object state. To support this, Java provides the Serialization facility.

Serialization provides the conversion of an object into a stream of bytes, the delivery of the byte stream through the network, and the object reconstruction on the other network end. At first glance, the serialization seems to be a lightweight operation, but it is not due to the object state portion, which could be very complex. Object code, on the other hand, presents no problem because it could be downloaded from a central point (e.g., an HTTP server, FTP server, email message, etc.), since it is static. The object state (data portion) must be handled at the moment of the serialization operation.

Serialization includes the marshaling of the object state (including all its inheritance), sending the object state through the network, and reconstructing the original object at the other end of the connection. To implement this, interpretation and buffering are required, but buffering is an extremely scarce resource on resource-constrained hardware platforms (for more information about Serialization, see the Java Object Serialization Specification [27]). Hence, it was decided to drop the moving of the object state, as there are no resources to handle it, and rely only on code mobility. Code mobility means that only the object code portion is moved (the .class file).

Another feature Jini introduces is location awareness. The services are spread across the Jini network or federation and there are mechanisms to find them and to invoke them, no matter where they are. This important feature is also supported during the Serialization operation since, besides the object state, it also includes the code base provider that "remembers" where the object code is stored. It seems that by removing the object state, the location awareness facility is also lost. This is true, but we have a very simple and lightweight solution to fix this. As our Java runtime only supports primitive types, there is no possibility to pass or receive objects from local or remote invocations. There is only one type of "object" that must be moved in the P2P Jini solution: the proxies of the P2P Jini services (application level). Since the object code portion alone is moveable, the location aware facility is handled during code download by saving the location of the target server (where the proxy must be connected when running) at the time the proxy is sent to the requester.

*5.2.3. Peer-to-Peer Jini Compliance with Ad Hoc Networks.* Sensor networks must be viewed as mobile ad hoc networks even in cases where some or all of the nodes remain stable. One reason for this is the battery-powered nature of these networks. When the battery is out of charge, nodes will die and the logical structure of the network changes. Due to this unstable/mobile characteristic of WSNs, we were also forced to make some adaptations to the traditional Jini architecture at the following levels:

(1) localization of the Lookup Service,

(2) localization of the Code Base Provider.

On traditional Jini systems, the Lookup Service(s) is assigned to one or more network nodes. This is acceptable because in stable infrastructure networks there is a higher level of network structure stability and availability. In the case of ad hoc networks, this critical Jini component must not be bound to a specific network node because, in case of node problems (crash, e.g.), all the Jini federation would stop working. To avoid this single point-of-failure, we argue that the Lookup Service must be spread across the entire network. This does not mean a distributed Lookup Service, but a Lookup Service per network node. A distributed Lookup Service would require management, join, and lease operations, while a Lookup Service per network node avoids these costly network operations. Considering the target hardware platform, the suppression of these operations also brings more savings in resources due to the fact that code and protocol interpretation are not required.

Similar to the Lookup Service, the Code Base Provider is a service that supplies the service code (.class file) to the Service Requester (data part is transferred via serialization). This duty makes Code Base Providers critical components for the proper functioning of Jini systems over ad hoc networks. For the same reasons that apply to the localization of the Lookup Service, we argue that each Service Provider must embed its own private Code Base Provider.

The two adaptations described above not only make the P2P Jini approach more lightweight to fit in target hardware platform but also bring other important advantages.

(a) As each sensor node exposes its own and private Lookup Service there is no need to implement Jini join protocol, Jini leasing mechanism, or Jini discovery protocol, either. This not only saves code

```
import java.net.*;
Import java.io.*;
public class JiniClient
{
   public static void main(String[] args) throws exception
   {
      tinyServiceRegistrar tSR=new tinyServiceRegistrar();
      String strProxyName=tSR.Lookup(new
      LusQueryItem("temp","near window", 4001);
      if(strProxyName.length() > 0)
      {
         Class tClass=Thread.currentThread().GetContextClassLoader().
               LoadClass(strProxyName);
         TempService t=(TempService)tClass.newInstance();
         int temp=t.GetTemp();
         //process read temperature here...
      }
   }
}
```

Algorithm 2: The P2P Client Application (SUN JVM).

```
public class PeerJiniClient
{
  public static void main()
  {
    char[] strProxyName=new char[10];
    char[] strServiceName=new char[]{"t", "e", "m", "p"};
    tinyServiceRegistrar tSR=new tinyServiceRegistrar();
    strProxyName=tSR.Lookup(new
        LusQueryItem(strServiceName,4,null,0,4001));
    if(strProxyName[0] != "\\u0000")
    {
        TempService x=(TempService) new RemoteServiceProxy();
        int temp=x.GetTemp(); //process read temperature here...
    }
  }
}
```

Algorithm 3: The P2P Client Application (JVM128).

and data memory at sensor nodes but also saves energy and network bandwidth.

(b) By implementing a Code Base Provider at each sensor node, we are easing the software development and deployment because there is no need for central Code Base Suppliers and, due to the removing of the data part of the "object mobility" it also eliminates the Code Base management. In the P2P Jini case, the Code Base Address is gracefully saved in each proxy at the time of the proxy downloading operation.

(c) The spreading of the critical Jini services (Lookup Service and Code Base Provider) across the network is a perfectly tailored solution for ad hoc networks because it brings robustness and availability to the entire network, even when several sensor nodes move or die.

Consider a test bed wireless sensor network with some sensor nodes (service providers) and a user with a hand-held device (service requester); also consider that the hand-held device issued a Lookup operation for a temperature service and a service invocation is imminent. Suddenly, the target sensor node dies or moves far way. When service invocation takes place, it fails because the target device does not exist anymore. This happens because the service proxy (requester side) cannot connect to service hosted by the dead device. If another service Lookup operation (for temperature service) is issued, nothing is returned because the device that hosts the temperature service is dead.

In standard Jini, due to delays and lease related times, a service proxy can be returned for a service that does not exist anymore. This happens because when a service proxy is registered in Lookup Service, there is a timestamp (lease time) that defines the validity of the service. In a scenario where the proxy is registered at time 0 and the lease time is specified for 1 minute and if the device dies 3 seconds after registering the service proxy, the Lookup Service will return the service proxy to the requesters for the rest of the 57 seconds. By returning the service proxy, the Lookup Service is saying that the service exists, which is false. This problem

```
public class OfferAndRequestService
{
  static short m_RemPort=0;
  static int m_proxyclassLen=0;
  static byte m_btState=0;
  static int proxyFileIndex=0;

  public static void main()
  {
    //This initial code is asynchronous as usual...
    //Sockets are configured and waiting asynchronously for connections
    int[] listenPorts = new int[2];
    listenPorts[0]=9930; //For broadcasts
    listenPorts[1]=4030; //For service invocation
    sockets.ServerSocket(2,listenPorts);

    //Now invoke the humidity service
    char[] strProxyName=new char[10];
    char[] strServiceName=new char[]{"h", "u", "m"};
    tinyServiceRegistrar tSR=new tinyServiceRegistrar();
    strProxyName=tSR.Lookup(new
    LusQueryItem(strServiceName,3,null,0,4001));
    if(strProxyName[0] != "\\u0000")
    {
      HumService x=(HumService) new RemoteServiceProxy();
      int hum=x.GetHum();
      //process read humidity here...
    }
  }
  //The rest of the source code is similar to the Algorithm 3.
   (...)
```

Algorithm 4: Sensor node offering and requesting services (JVM128).

could be bigger if the lease time is defined as "forever". On the other hand, specifying short lease times (e.g., 2 or 5 seconds) is not a solution because more data exchange is required between the device that hosts the Lookup Service and the device that hosts the service.

Suppose now that a new sensor node joins the network and offers the humidity service. From the P2P Jini point of view, this new service becomes available at the same instant the node joins the network (and it becomes unavailable at the same instant the node leaves network) and any P2P Jini requester can find it. This happens because in P2P Jini it is the node that hosts the service that answers to service requests.

Unlike P2P Jini, in standard Jini before the new service is available to the network, the device hosting the service must firstly discover an available Lookup Server, then register service proxy, and then establish the related lease time. The service is only available after this setup operation.

Hence, P2P Jini can assure an up-to-date view of the service network without any cost.

*5.2.4. Adaptation and Supression of Standard Jini Protocols.* In traditional Jini systems, the Lookup Service stores proxy objects. In P2P Jini a Lookup Service, called tinyLUS, stores service descriptions plus the name of the local class file that implements the service proxy (see Figure 3). Service descriptions include similar data structures as a Name Java object plus a Comment Java object (more data could be easily added). Each node also includes a proxy service similar to a standard Registrar Jini proxy to interact with Lookup Services. In the P2P Jini case, the proxy for the Lookup

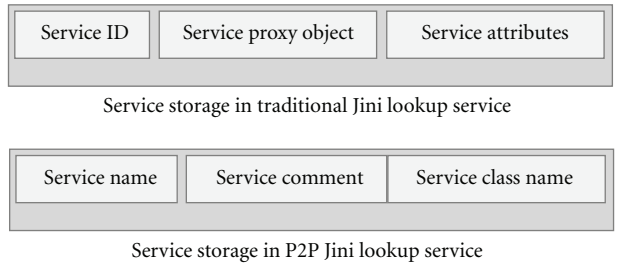


Figure 3: Contents of the Jini Lookup Service versus P2P Jini Lookup Service.

Service was named tinyServiceRegistrar. All the transformations made to the standard Jini had a positive impact on the standard Jini protocols. The discovery protocol which allows locating Lookup Services is not needed anymore because the Lookup Service will exist in the whole network (one per node). The join protocol, which enables a Jini Provider to register its services in the Lookup Services, is not needed either, because in P2P Jini each device registers its services in its local and private Lookup Service (tinyLUS). The lookup protocol, which enables a requester to locate services on Lookup Services, was modified to receive service descriptions and comments, while standard Jini requesters look for services by using their Service IDs, service details, or service interfaces.

In P2P Jini, the Lookup operation first searches in the local Lookup Service and then on all the other remote Lookup Services by the use of (natural) broadcasting. During the Lookup operation, the P2P Jini Requester also informs

the target tinyLUS about its listening port used to receive the service proxy. Every tinyLUS that has the service being looked for connects to the requester's port listener in order to send back the service proxy. This is done after tinyLUS waits for a random time in order to avoid a reply storm. By proceeding this way, only one tinyLUS connects to the Service Requester because the other ones, which also match the requested service, will see the communication channel occupied and will give up.

## 6. Evaluation

This section aims to evaluate the service discovery and service usage facilities provided by P2P Jini and to show the ease of writing applications, while consuming little memory, both in ROM and RAM. To achieve that, both sensor nodes and a desktop machine request and offer services. The source code presented next was executed on STK300 evaluation boards behaving as ATMega128 L-based wireless sensor nodes.

*6.1. Sensor Node as a P2P Jini Service Provider.* In this evaluation test case, the sensor nodes were programmed as P2P Jini Service Providers. For simplicity, each node just exposes one service (e.g., temperature). The support for more than one service per sensor node does not pose any difficulty because it is simply a new service unique identifier that is tested during the dispatch of the service request. As expected, this Java application must define an application, a Lookup Service, a Code Base Provider, and the effective P2P Jini Service being offered. The P2P Jini service was registered with the "temp" name and the "near window"' comment.

The service proxy was not referenced because it was created using a simple software utility. In the future, all the basic functionality of the P2P Jini software components could be created by a P2P Jini development framework. Nevertheless, the service proxy class file for the temperature service occupies about 1.3 KB.

The Algorithm 1 Java code implements the local Lookup Service (sensitive to broadcast requests), the Code Base Provider, and the Temperature Service as well. It can also be seen that the socket class was implemented by using synchronous callbacks. As Algorithm 1 shows, to transform a sensor node into a collection of services there is the need to write hundreds of Java source code. This source codes is written once and then burned on sensor nodes ROM. Once again, all the source code can be easily generated and compiled by a framework (as it follows a template structure), freeing the programmer from writing a low-level server Java code.

Algorithm 2 shows the simplicity of writing a P2P Jini client application. Generically, a client application must instantiate the proxy that aids in service lookup (tinyServiceRegistrar), issue a lookup operation for a service, and then issue the service invocation (of course, like in standard Jini, the client must know the interface implemented by the service proxy). As already stated before, P2P Jini uses a message-oriented protocol for method invocation. Although low-level, it is completely transparent to the P2P Jini client due to the use of service proxy.

Table 1: Memory Usage for P2P Jini Server at Sensor Node.

| JVM128 | Space required (bytes) |
|---|---|
| ROM usage for firmware (VM) | 34102 |
| ROM usage for bytecode storage | 3646 |
| Total ROM usage | 37748 |
| RAM usage for firmware (VM) | 893 |
| RAM usage for bytecode storage | 0 |
| Total RAM usage | 893 |
| Available heap | 2403 |
| Maximum heap usage | 842 |

Table 1 shows the sensor node memory footprint when the temperature service is offered by the sensor node. The download of the temperature service proxy took about 4 seconds and the service, request took less than 1 second. These delays are common to all evaluation scenarios.

The table above clearly shows that the P2P Jini approach is perfectly suitable for the target hardware. Even when executing a Lookup Service, a Code Base Provider, the P2P Jini Temperature Service and a Java virtual machine, more than half of the heap memory is still available.

*6.2. Sensor Node as a P2P Jini Requester.* To the best of our knowledge, no other SOA solution for resource constrained devices offers the requester actor. For instance, the CMatos commercial and patented solution can transform a limited device into a 100% compliant Jini server, but that device cannot request services from other devices. This is a clear limitation on flexibility.

Algorithm 3 shows how simply a resource constrained device could be transformed into a P2P Jini requester. In this case, the P2P Jini server can be in a PC, a sensor node, or in any other device in the network.

The sensor node code starts by instantiating the local tinyServiceRegistrar class in order to launch a service lookup. The lookup method of the tinyServiceRegistrar class sends a broadcast message and waits for requests on the 4001 TCP port. If some service is found, then the same method receives the service proxy and stores it in EEPROM memory. Then, the proxy is loaded and cast on the proper interface and the method is invoked. The RemoteServiceProxy class is used to load the proxy that was received after issuing the lookup operation. Table 2 presents the sensor nodes memory impact of the P2P Jini client application.

According to Table 2, the P2P Jini requester role uses more RAM than a P2P Jini server role. This is due to the instantiation of the tinyServiceRegistrar class. A static version of the lookup method, which belongs to that Java class, would save RAM on the device.

*6.3. Sensor Node Offering and Requesting P2P Jini Services.* This third test case represents a scenario where a specific sensor node (Node A) needs "cooperation" from other(s) node(s) and will request services from them. Node A will

Table 2: Memory Usage for P2P Jini Client Application.

| JVM128 | Space required (bytes) |
|---|---|
| ROM usage for firmware (VM) | 34102 |
| ROM usage for bytecode storage | 4246 |
| Total ROM usage | 38348 |
| RAM usage for firmware (VM) | 893 |
| RAM usage for bytecode storage | 0 |
| Total RAM usage | 893 |
| Available heap | 2403 |
| Maximum heap usage | 997 |

Table 3: Memory Usage for P2P Jini Client and Server Application.

| JVM128 | Space required (bytes) |
|---|---|
| ROM usage for firmware (VM) | 34102 |
| ROM usage for bytecode storage | 4846 |
| Total ROM usage | 38948 |
| RAM usage for firmware (VM) | 893 |
| RAM usage for bytecode storage | 0 |
| Total RAM usage | 893 |
| Available heap | 2403 |
| Maximum heap usage | 1042 |

also offer services to the other nodes. As Algorithm 4 shows, node A will be programmed to offer the temperature service and to request the humidity service from nodes surrounding it. To achieve that, the Java source code is very similar to Algorithm 1. The difference will reside in the main function, which must define sockets for service offering and, then, must search for a humidity service and invoke it. This test case is also used to prove that even resource-constrained sensor nodes like MICA2 can offer and request P2P Jini services without relying on external devices. To the best of our knowledge, this is a unique characteristic of the P2P Jini solution.

In Algorithm 4, the main function starts by defining the needed sockets to receive service requests (for the temperature service) and then issues a lookup for an available humidity service. Once socket support of the JVM128 Java Virtual Machine is fully asynchronous, the waiting for service requests does not forbid the main function from requesting services.

According to Table 3, a resource-constrained sensor node is capable of offering and requesting services at the same time, because it will not compromise its available resources. This challenging application uses less than a half of the available RAM (heap).

## 7. Related Work

P2P Jini is not the first service-oriented middleware able to program WSNs. Both commercial and academics solutions exist. Arch Rock PhyNet [14], Atlas [15], and CMatos [28] are examples of commercial solutions but none of them support the requester and supplier roles without relying on external resource rich devices. However, this section details only the works done on academia.

UbiSOA [29] is a service-oriented architecture that allows programmers to integrate RFID tags and wireless sensor networks, through the abstraction of Web Services in a centralized model. The gateway provides two kinds of Web Services to the outside world: network services and RFID services. Between gateway and sensor network and RFID a private protocol is used. Unlike P2P Jini, UbiSOA cannot be used in sink-less emergency scenarios and it is not flexible enough to be used in ubiquitous spaces with spontaneous networking because there is only the service provider role.

USEME [30], a service-oriented framework for wireless sensor and actor networks, tries to give the programmer the power to define applications, groups, events, and services through an abstract language. Then, the framework converts the abstract language to Java, C#, NesC, and so forth. The authors argue that a sensor node can be programmed to offer and request services. The project seems to be in its beginning because authors refer nothing about how to write an application, how to invoke a service, and how to subscribe events and no proof of concept is presented. It is not clear if it follows a centralized or distributed model. This proposal will require a big effort from the programmer in designing applications (defining groups, services, nodes, events, etc.) and the programmer must learn a new programming language in order to use this solution. Unlike USEME, P2P Jini follows a pure and well-defined service-oriented approach, based on well-known interfaces, and offers a widely accepted programming language and programming paradigm, while standard programming tools can be used (e.g., javac, eclipse, NetBeans, etc).

In [31], a demo is presented, which implements a multilevel SOA-based architecture for heterogeneous sensor networks which follows the centralized model. This proposal segments available devices into three categories, for example, full capacity nodes (servers, desktops, and laptops), limited capacity nodes (hand held devices), and low-capacity nodes (sensor networks) and proposes three different SOA approaches to address the different types of devices. Full capacity nodes use standard Web Services, limited capacity nodes use Devices Profile for Web Services (DPWSs) [32], and for low-capacity nodes authors implemented the wsnSOA. WsnSOA implements a low-level and private protocol based on TinyOS messages. Low-capacity nodes are prepared to send service announcements or answer to service announcement requests and the services are consumed in a publish/subscribe model in order to save resources. P2P Jini differentiates from this proposal in three main aspects: firstly, P2P Jini, through the use of service proxies, uses a unique SOA approach for all devices. This eliminates the use of bridges and requires less configuration and setup time. Secondly, P2P Jini sensor nodes offer real software services, while in this demo project, the sensor nodes are addressed through the network gateway. Thirdly, P2P Jini allows sensor node cooperation in a service-oriented model, while in this demo, sensor nodes cannot cooperate with each other.

MiSense [33] is another service-oriented solution, which follows the centralized model. Sensor nodes send service

description messages (TinyOS messages) to the gateway in order to fill the service repository and applications use the publish/subscribe model in order to consume network services. Authors do not define which SOA standard is used between applications and the network gateway. Again, this proposal relies on TinyOS and NesC programming language (high-level curve), while P2P Jini relies on Java programming language. This proposal does not allow node cooperation and requires a big effort at the time of system deployment.

TinySOA [18] is very similar to MiSense, as it follows the centralized model and sensor nodes register their capabilities in network gateway, through the use of TinyOS messages. Then, any application can use those capabilities (services) by using Web Services approach.

Finally, Extended Service-Oriented Architecture (ESOA) [9] is the most recent related work. In this work, the authors added a service composition layer to the top of the service-oriented architecture and everything runs above the LiteOS operating system. However, due to lack of published information, no characterization or comparison can be done.

## 8. Conclusion

This article presents a new SOA approach, called P2P Jini, to address wireless sensor networks in a perspective of services. It fits in the most resource-constrained sensor nodes (such as the MICA2 platform) and is able to transform those resource-constrained sensor nodes into real services; that is, each node is addressed by the services it offers. This fact constitutes an extremely important abstraction for programmers because it helps them focus on the application and not on low-level network details.

A P2P Jini sensor node can also search and use P2P Jini services in other nodes. As sensor nodes can have the flexibility of offering and requesting services, they become more intelligent. As P2P Jini can fit virtually on any computing device (from embedded ones to servers), it can be used to create smart spaces where federations of devices communicate and cooperate among themselves, in order to bring more convenience to life (e.g., a smart home). However, this is just a first proof of concept. In the future, a publish/subscribe mechanism will be included and the communication device will work in a TDMA approach, in order to save energy.

## References


[1] T. Schmid, H. Dubois-Ferriere, and M. Vetterli, "Sensorscope: experiences with a wireless building monitoring sensor network," in *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN '05)*, June 2005.

[2] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[3] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proceedings of the 2nd International Conference on Mobile Data Management (MDM '01)*, 2001.

[4] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TAG: a tiny aggregation service for ad hoc sensor networks," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[5] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy, "Rethinking data management for storage-centric sensor networks," in *Proceedings of the 3rd Biennial Conference Innovative Data Systems Research (CIDR '07)*, January 2007.

[6] M. Ameen, S. Islam, and K. Kwak, "Energy saving mechanisms for MAC protocols in wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2010, Article ID 163413, 16 pages, 2010.

[7] H. Sabbineni and K. Chakrabarty, "An energy-efficient data delivery scheme for delay-sensitive traffic in wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2010, Article ID 792068, 14 pages, 2010.

[8] M. Weiser and S. Brown, "The coming age of calm technology," Tech. Rep., Xerox PARC, 1996.

[9] V. Vanitha, V. Palanisamy, N. Johnson, and G. Aravindhbabu, "LiteOS based extended service oriented architecture for wireless sensor networks," *International Jounal of Computer and Electrical Engineering*, vol. 2, no. 3, 2010.

[10] S. Shin, "Jini™ surrogate architecture," Sun Microsystems, 2001.

[11] P. Huang, V. Lenders, P. Minnig, and M. Widmer, "Jini for ubiquitous devices," ETH TIK-Nr. 137, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, 2002.

[12] S. Deter and K. Sohr, "Pini—a Jini-like plug & play technology for the KVM/CLDC," in *Proceedings of the Innovative Internet Computing Systems (IICS '01)*, LNCS 2060, pp. 53–67, Springer, 2001.

[13] T. Hasu, *Research seminar on telecommunications software*, M.S. thesis, Telecommunications Software and Multimedia Laboratory at Helsinki University of Technology, 2002.

[14] Arch Rock web site, May 2010, http://www.archrock.com.

[15] J. King, R. Bose, S. Pickles, A. Helal, S. Ploeg, and J. Russo, "Atlas: a service oriented sesnor platform," in *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN '06)*, pp. 630–638, 2006.

[16] J. Waldo, "The Jini architecture for network-centric computing," *Communications of the ACM*, vol. 42, no. 10, pp. 76–82, 1999.

[17] N. Costa, A. Pereira, and C. Serôdio, "A Java software stack for resource poor sensor nodes: towards peer-to-peer Jini," in *Proceedings of the 4th International Conference on Embedded and Multimedia Computing (EM-COM '09)*, pp. 114–119, December 2009.

[18] J. A. García-Macías and E. Avilés-López, "TinySOA: a service-oriented architecture for wireless sensor networks," *Service Oriented Computing and Applications*, vol. 3, no. 2, pp. 99–108, 2009.

[19] The leJOS web site, December 2009, http://lejos.sourceforge.net.

[20] The TinyVM web site, December 2009, http://tinyvm.sourceforge.net.

[21] The Wabasoft web site, December 2009, http://www.wabasoft.com.

[22] CrossBow web site, December 2009, http://www.xbow.com.

[23] The Kanda web site, December 2009, http://www.kanda.com.

[24] The Sparkfun Electronics web site, December 2009, http://www.sparkfun.com.

[25] V. Lenders, P. Huang, and M. Muheim, "Hybrid Jini for limited devices," in *Proceedings of the IEEE International Conference on Wireless LANs and Home Networks*, pp. 27–34, 2001.

[26] The kXML-RPC web site, December 2009, kxmlrpc.objectweb.org.

[27] Java Object Serialization Specification, August 2009, http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html.

[28] The Psinaptic Inc. web site, February 2009, http://www.psinaptic.com.

[29] J. García-Macías and E. Avilés-López, "Developing ubiquitous applications through service-oriented abstractions," in *Proceedings of the 3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008*, vol. 51 of *Advances in Soft Computing*, 2009.

[30] E. Ca, J. Chen, and L. Llopis, "USEME: a service-oriented framework for wireless sensor and actor networks," in *Proceedings of the 8th International Workshop on Applications and Services in Wireless Networks (ASWN '08)*, pp. 47–53, 2008.

[31] J. Leguay, M. Lopez-Ramos, K. Jean-Marie, and V. Conan, "Service oriented architecture for heterogeneous and dynamic sensor networks," in *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS '08)*, pp. 309–312, July 2008.

[32] "Devices Profile for Web Services," January 2011, http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html.

[33] K. Khedo and R. Subramanian, "A service-oriented component-based middleware architecture for wireless sensor networks," *International Journal of Computer Science and Network Security*, vol. 9, no. 3, 2009.