# WSN-Projects: Lecture-3

JVM & TakaTuka Introduction

# Contents

- Java advantages
- Java limitations
- JVM Responsibilities
  - Loader
  - Verifier & initialization
  - Interpreter or JIT-compiler
  - Garbage collection
  - Thread scheduling
- JVM concepts
  - Classfile format
  - Constant pool
  - Bytecode
  - Data structures

# Contents

- TakaTuka design and optimization
  - SUN Squawk design
  - TakaTuka design and goals
  - TakaTuka CP optimization
  - TakaTuka bytecode optimization
- Your project

# Java advantages

- Java is object oriented
  - Good and clean design
  - Easy to modify and extend
  - Easy to understand
  - Unlike C++ it is fully object oriented (minus native methods calls)
- Ease of use and learn
  - First language in many universities
  - High level concepts
- Big community
  - Java has a very big community
  - Java code itself is now open-source from Sun
  - Many tools, IDE are available

# Java advantages

- Memory protection
  - No dangling pointers
  - No segmentation fault errors
  - Automatic garbage collection
- Portability
  - Same set of binaries executable on different platform
    - Program once run everywhere
  - Why its portable?
  - For motes provide options of partially updating a project using *over the air programming*
    - Change few class files instead of whole big application

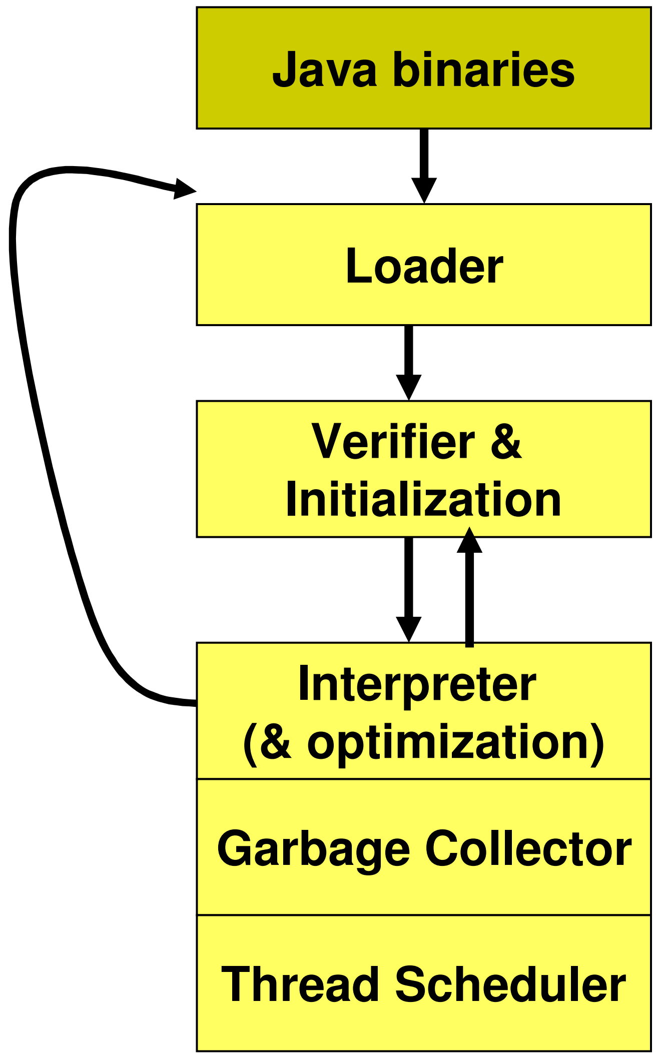
# Java limitations

- Java is slow
  - Especially with interpreter and not much with JIT-compiler
- Java is big on disk
  - Class files are bigger than executable of C
- Java is big in RAM
  - Java takes lots of RAM

# JVM responsibilites

University of Freiburg
Institute of Computer Science
Computer Networks and Telematics
Prof . Christian Schindelhauer

# JVM Responsibilities




(also known as linking)

# Loader

- Find the class binaries and bring it to JVM
- When to load (classically)
  - Loading is dynamic
  - When a class is first time accessed
    - e.g. new opcode is called or some static field is accessed etc

# verification & initialization

- When? (classically)
  - After loading a class
  - Dynamic as loading
  - Some verification are perform also during .java to .class conversion
    - They are usually repeated during class loading and accessing
- What?
  - *Verification* checks that the binary representation of a class or interface is structurally correct
    - Examples: (1) operands are valid, (2) branch address are valid, (3) method signatures and method call matches, (4) private field is not access outside a class etc.
- Initialization
  - Execute static initializers and initialize the static fields

# Interpreter or JIT-compiler

- Java binaries (class-files) are machine independent
  - Make Java portable
- Interpreter Vs Just-In Time (JIT) Compiler
  - Interpreter interpret one instruction at a time
  - JIT compiler make most frequent code parts into machine code (e.g. loops) so that they can run faster

# Interpreter or JIT-compiler

- Why interpreter for mica2?
  - Mica2 has Harvard architecture
  - In Harvard architecture program and data memory are physically separated
    - Flash: Program memory
    - RAM: Data memory
  - In order for JIT to work one has to write machine-code generated into the Flash
    - Writing in the flash is slow
    - Flash can be written in finite number of times
      - Each program may has to write hundreds of time during its execution
  - Interpreter
    - Execute directly from Flash without any dynamic machine dependent code generated

# Interpreter or JIT-compiler

- TakaTuka Aim
  - Make interpreter run faster based on recent research
  - Make it light weight for RAM

# Garbage collection

- Garbage collection clears the memory for future use
- Unlike C, programmer do not have to worry about freeing memory
- When and how to do garbage collection
  - Nothing specific
  - Each JVM can handle it the way it prefers
- TakaTuka want to have
  - Real time garbage collection
  - A light garbage collection with no significant memory usage
  - A garbage collection with small CPU usage
    - Conserve battery lifetime

# Thread Scheduler

- Schedules what next thread to run
- May interrupt currently running thread for schedule a higher priority thread
- Threads are much better as compared to Event-driven model (TinyOS)
  - But have significant memory overhead
- TinyOS Aims
  - Scheduler that consume very small memory
    - Current implementation is partially in Java and hence consumes RAM
  - Threading model that use single stack for all the thread

# JVM concepts

# Class file format

```
ClassFile {
    u4 magic;                              // it is always 0xcafebabe
    u2 minor_version;       //minor and major version tell class-file format version
    u2 major_version;
    u2 constant_pool_count;                //size of the constant pool
    cp_info constant_pool [constant_pool_count-1];  //the constant pool (CP)
    u2 access_flags;                       //tells if the class is public, private etc
    u2 this_class;              //this class pointer (points to class name in the CP)
    u2 super_class;                        //super class pointer (note that only one)
    u2 interfaces_count;                   //number of interfaces
    u2 interfaces [interfaces_count];      //interfaces (CP entry of class_info)
    u2 fields_count;                       //number of fields
    field_info fields[fields_count];       //fields_info array
    u2 methods_count;                      //field count
    method_info methods[methods_count];              //method_info array
    u2 attributes_count;                   //attribute count
    attribute_info attributes[attributes_count];    //attribute_info array
}
```

# Constant Pool

- A CP of a class is a collection of distinct constant values of variable size
- It reduces the size of Class file
  - Each constant appears once in constant pool
  - A constant usually is larger than two bytes
  - Constant in constant pools are referred using two bytes
  - A constant could be referred multiple times
- Example
  - "Hello World" takes 11 bytes
  - Let say it is used five times in a special class file format without a CP
    - Total number of bytes used 11*5 = 55
  - In normal class file with CP
    - Total number of bytes used 11+(2*5) = 22

# Bytecode

- Method_info has the bytecode array (in the code_attribute)
- Bytecode has a set of *instruction*
- Instruction
  - One byte op-code
  - Zero to many byte operand
- Op-code
  - Tells what the instruction is
  - One byte opcode in Java but only 204 instructions
- Operand
  - Any input for the instruction
  - Not all instructions have no operands
- Example
  - iload 5 //load local variable #5 on the operand stack
    - Opcode (Mnemonic form) is “iload”
    - 5 is the operand

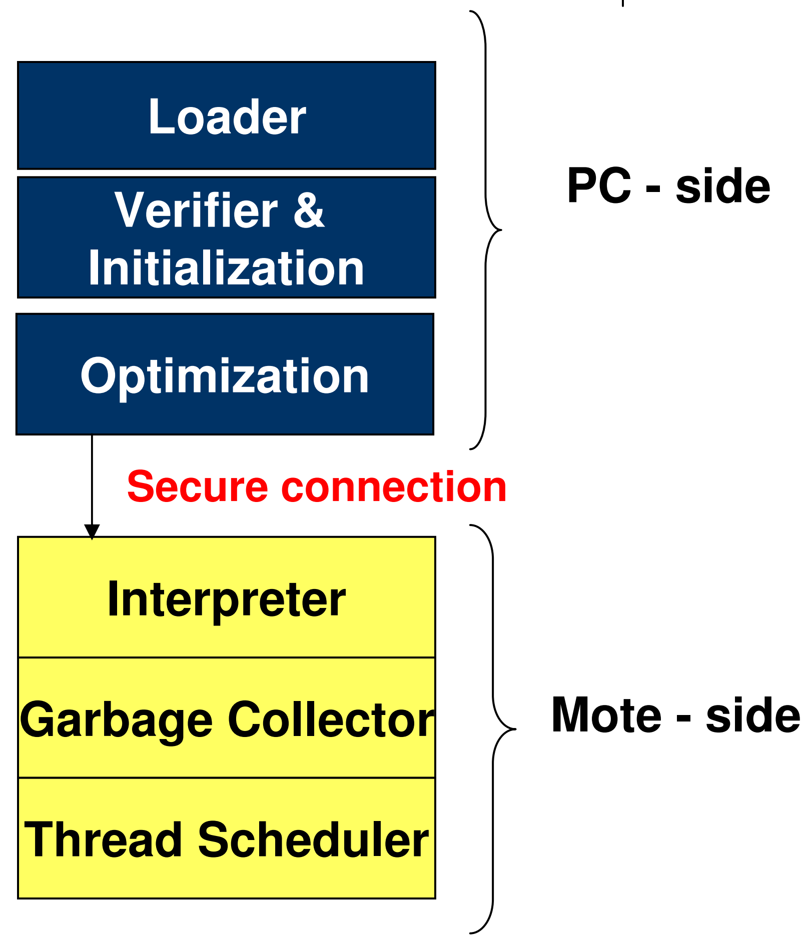
# Data structures

- Stack
  - Java has stack based instruction set
    - Instead of a register base instruction set
  - Many instruction either *push* or *pop* from operand stack
  - For example *iAdd* will add two integers on the top of stack
    - These values could be added by a function return or *iload* or some other way
  - The operand stack of a function depends on the instructions sequence of that functions
- Heap
  - Heap is the place where object are placed
  - Heap size depends upon the number of objects in the memory and their local variables

# TakaTuka design and optimization

# Squawk design

- Split VM architecture
  - Some part of JVM are perform on PC
  - Rest on the mote
- Advantages
  - Split JVM run faster
  - Avoid memory consuming tasks
  - Less run-time errors
- CLDC Compliance?
  - Code must be verified and secure
  - To make sure
    - PC to mote data transfer must be through secure connection



# TakaTuka design & goals

- Based on Squawk split-VM-architecture
- Better and more aggressive code optimization as compared to Squawk
- To make Java by > 95% smaller on disk and RAM
- To make it run much faster with an interpreter
  - Motes sometimes cannot have JIT-compiler due to Flash limitation

# TakaTuka CP Optimization

University of Freiburg
Institute of Computer Science
Computer Networks and Telematics
Prof. Christian Schindelhauer

- Traditional: Duplicate values in the project of per class CP
  - Class A has “Hello World” in its CP
  - Class B has “Hello World” in its CP
- TakaTuka: Global CP per project
  - Class A and Class B now have single constant pool with one “Hello World”
- Traditional: Numeric types in CP has fixed length
  - Long will always be 8 bytes and integer/short/boolean always 4 bytes
  - Example long l = 5; will take 8 bytes in the CP
  - Example static final short s = 7; will take 4 bytes in CP
- TakaTuka: Numeric types are variable length
  - Example long l = 5; will take now only 1 byte
  - Example static final short s = 7; will take one byte in CP

# TakaTuka CP optimization

- Traditional: Names resolution information
  - Class, functions, fields names are written in the CP
  - They are use for dynamic loading and debugging
- TakaTuka: Name resolution information
  - Pre-loading on PC as in split-VM-architecture
  - No naming information are transferred to motes

# TakaTuka bytecode optimization

University of Freiburg
Institute of Computer Science
Computer Networks and Telematics
Prof. Christian Schindelhauer

- All bytecodes space is not used
  - Java has 204 opcodes hence 52 are not used
  - A program not use all 204 opcodes
- TakaTuka
  - Use available opcodes (not used by a program)
  - To create new instructions
  - Objective is increase speed
  - Reduce size

# TakaTuka bytecode optimization

University of Freiburg
Institute of Computer Science
Computer Networks and Telematics
Prof. Christian Schindelhauer

- TakaTuka: Two types of bytecode optimization
  - Single instruction optimization
  - Multiple instruction optimization
- In summary
  - We reduce the size of existing single instructions
  - Combine existing instruction
  - Use all of opcodes not used by a program to make such new instructions
- Why increase speed
  - Obvious that size decreases
  - Speed increase because less number of *instruction dispatch* is required

# Your projects

- Five Projects
  1. **Dead code removal** – pure java project
     - Extendable to Bachelor thesis
     - Group leader:
  2. **Multi-threading** – mostly C project
     - Extendable to Bachelor thesis
     - Group leader:
  3. **10 Tiny Projects** – mostly C project
     - Excellent start for a Bachelor thesis about Garbage collection or CLDC compliance
     - Group leader:

# Your project

4. **TinyOS integration --- TinyOS/NesC, Java and C**
   - Extendable to Bachelor thesis
   - Group leader:
5. **Garbage collection** – C and some Java
   - Extendable to Bachelor thesis
   - Group leader:

# References

- Lindholm et al. "*The JavaTM Virtual Machine Specification*", Second Edition

# The End

- Thank you for listening