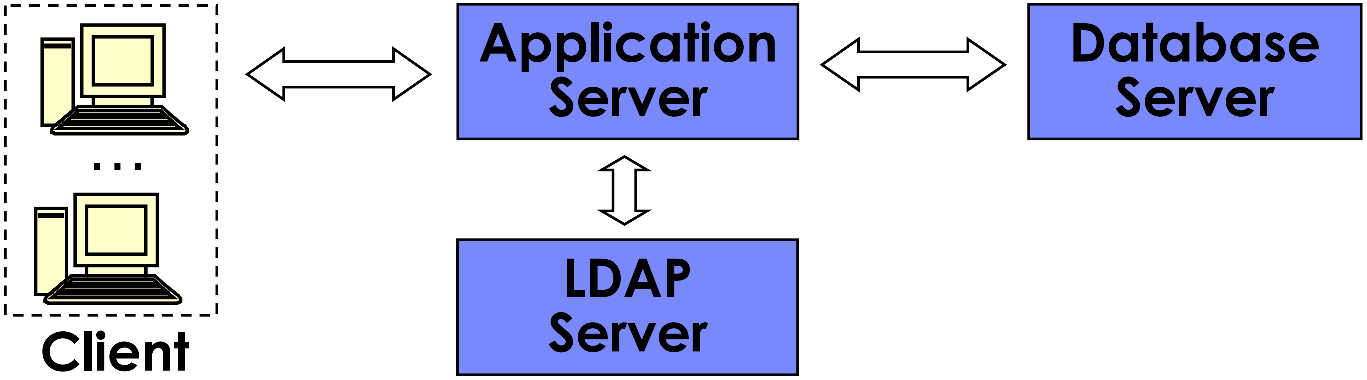
Using the Middle Tier to Understand Cross-Tier Delay in a Multi-tier Application

Haichuan Wang(1), Qiming Teng(1),
Xiao Zhong(1), Peter F. Sweneey(2)
(1) IBM Research – China, {wanghaic,tengqim,zhongx}@cn.ibm.com
(2) IBM Watson Research Center, pfs@cn.ibm.com

Motivation

- Enterprise applications have multi-tier architectures
- A performance bottleneck on any tier may cause the whole system to under perform



Application Server

Database Server

LDAP Server

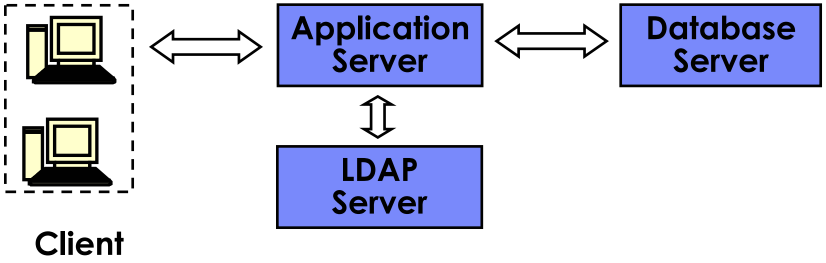
Client

Observed:
Low throughput
Low CPU utilization

?

Which tier causes the bottleneck?

Previous Approaches

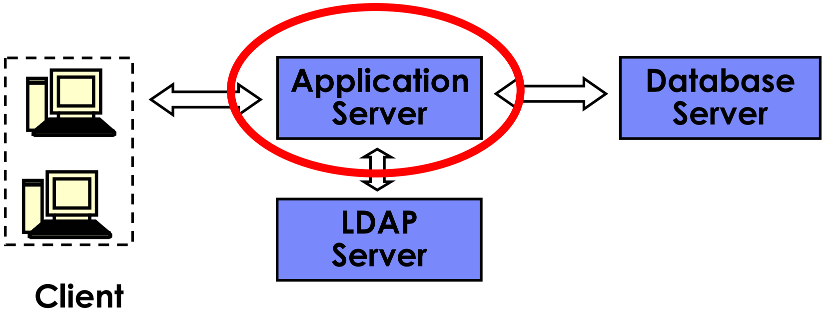


1. Collect system metrics on all tiers
   - Statistics on each machine
2. Aggregate resource consumption
   - Interaction between machines
3. Build a whole system interaction model

- Some Limitations
  - Hard to collect system metrics on all tiers in some production systems
    - e.g. Thousands of clients; out-bound servers
  - Complex
  - Identify performance bottlenecks based on a large number of metrics

Proposed Approach – Focus on the Middle-tier

- Focus on the middle-tier
  - Application server (Java based)



- Track cross-tier method invocations in Java level
  - Identify method invocations that handle cross-tier interactions
  - Extract "contextual information" associated with these method invocations
- Identify the blocking in native level
  - Trace thread interruptible (blocking) state
  - Map back to the cross-tier method invocations
- Refer to the blocking source tier
  - Based on the contextual information

Example

SocketRead(…)

↓

Call Stack: LDAP Connector
IP Address: xxx → LDAP Server

↓

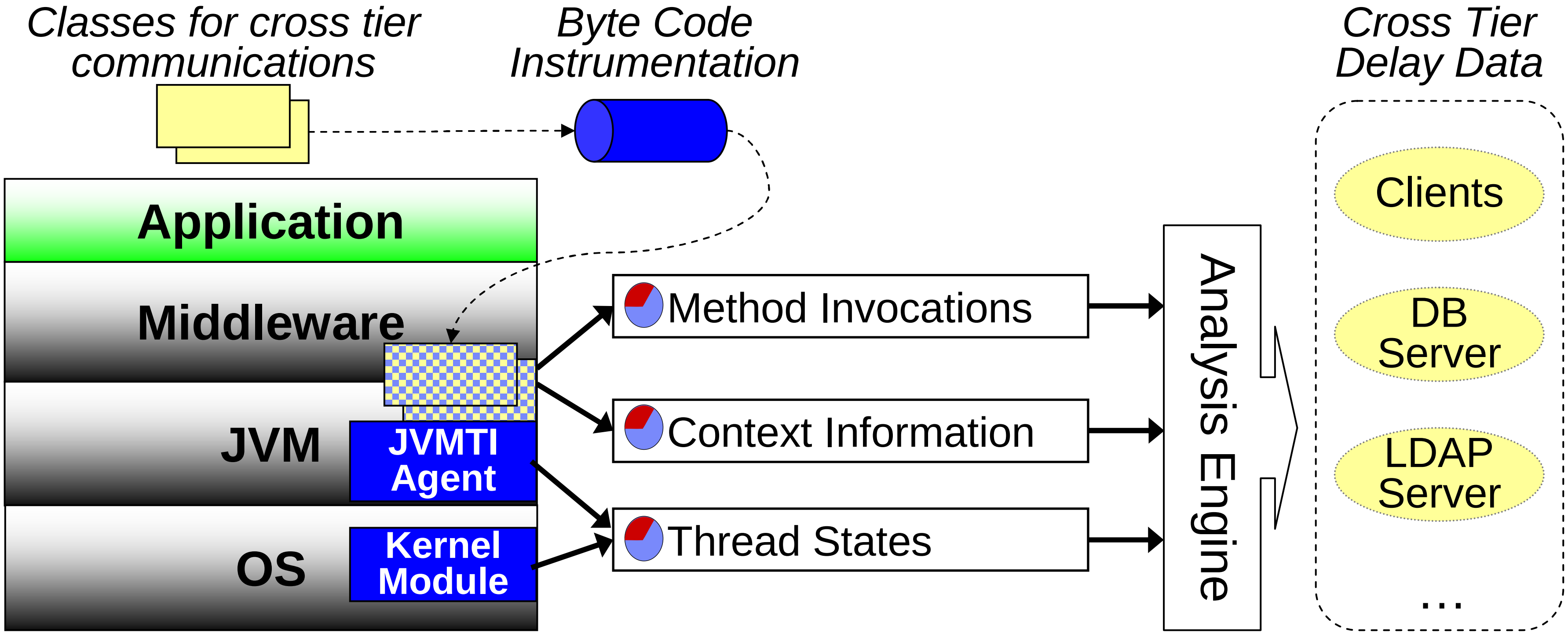
Thread Blocking State:
SocketRead(…): Blocked

↓

Tier of Blocking Source:
LDAP server (xxx)

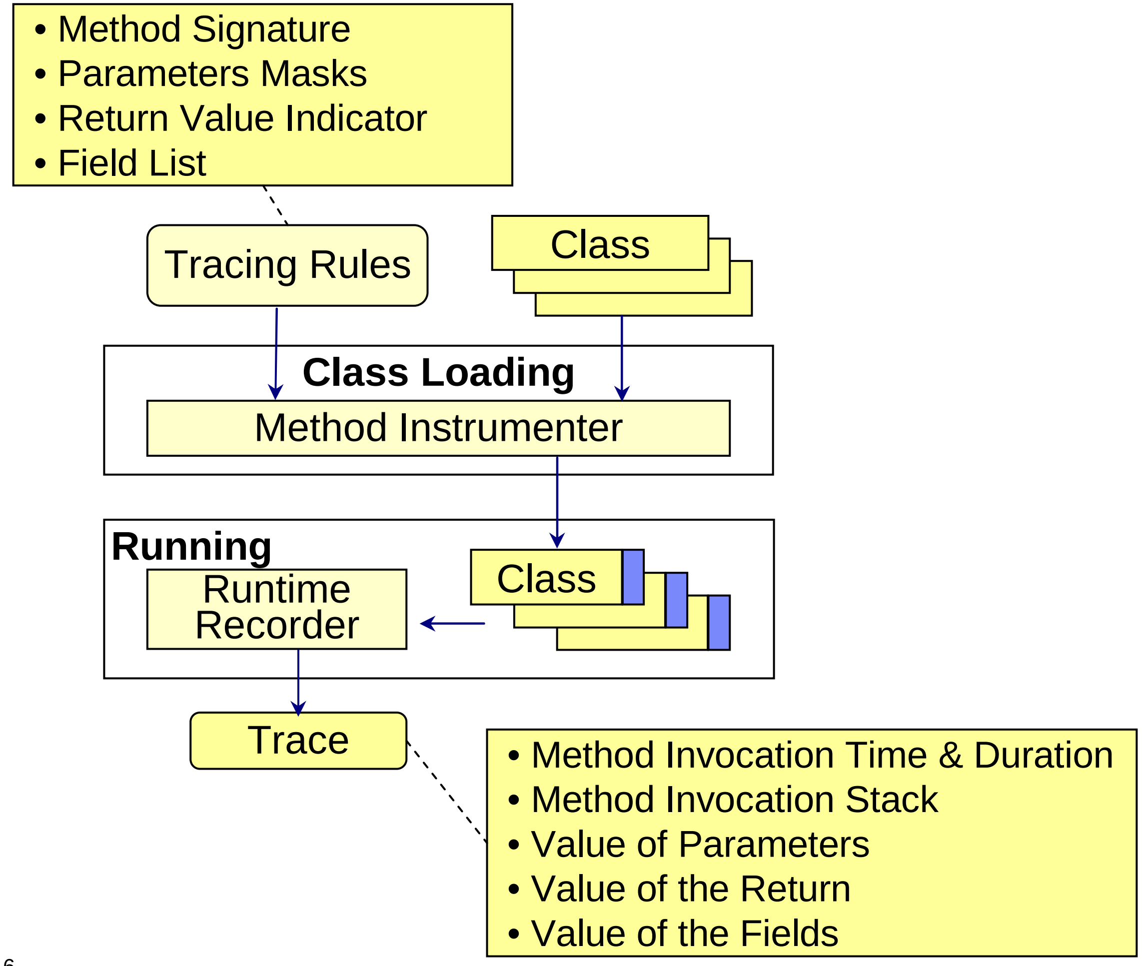
Solution Architecture Overview

- Constructing Cross-Tier Delay data from the following data
  - –Method invocation by dynamically byte code instrumentation
  - –Context information by dynamically instrumentation
  - –Thread States by JVMTI agent and a kernel module



Tracing Method Invocations – Class Instrumentation

- Tracing rules driven Java byte code instrumentation



- Method Signature
- Parameters Masks
- Return Value Indicator
- Field List

Tracing Rules

Class

Class Loading

Method Instrumenter

Running

Runtime Recorder

Class

Trace

- Method Invocation Time & Duration
- Method Invocation Stack
- Value of Parameters
- Value of the Return
- Value of the Fields

Example Tracing Rules

Method:
java.io.InputStream java.net.PlainSocketImpl.getInputStream();

Parameters Mask:

Record Return: False

Fields:
java.net.InetAddress address;
java.io.FileDescriptor fd;

Method:
int java.net.SocketInputStream.socketRead0(**java.io.FileDescriptor para0**, byte[] para1, int para2, int para3, int para4);

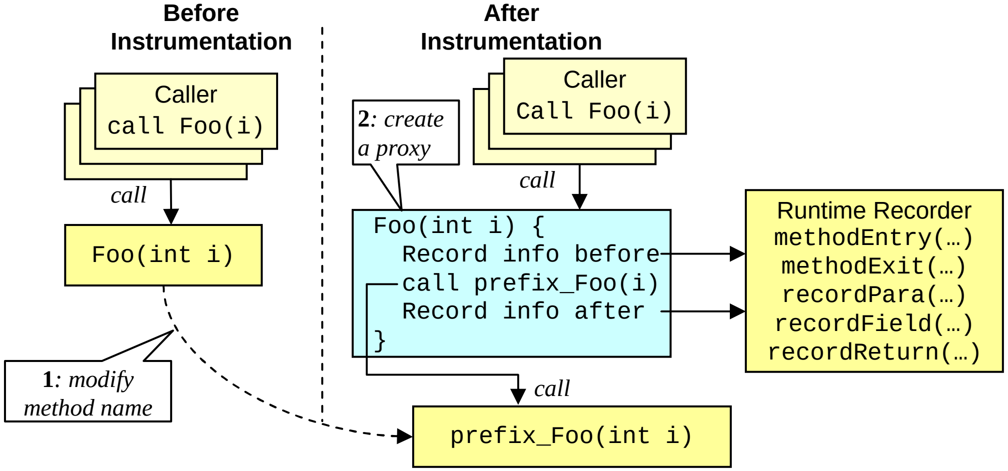
Parameters Mask: 10000

Record Return: True

Fields:

Tracing Method Invocations – Class Instrumentation (2)

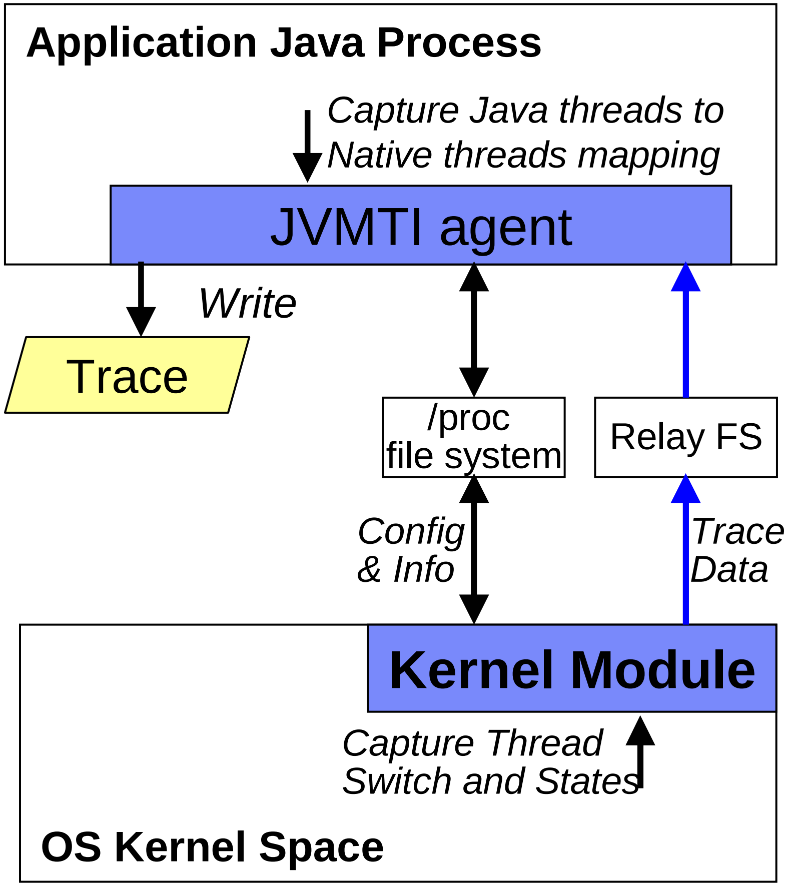
- Three different approaches for dynamically instrumenting methods
  - Create proxy methods



  - Directly instrument the prolog and epilog of an identified method
    - In case we cannot insert the proxy
  - Instrument all call-sites of an identified method
    - For tracing “JNI” methods in the JVM without the JNI prefix mechanism

Trace Thread Blocking

- Kernel Module
  - Based on Kprobe (Linux)
  - Inserted into OS scheduler
  - Only collect thread interruptible native states (blocked)
- JVMTI Agent
  - Assist to map native threads to corresponding Java threads

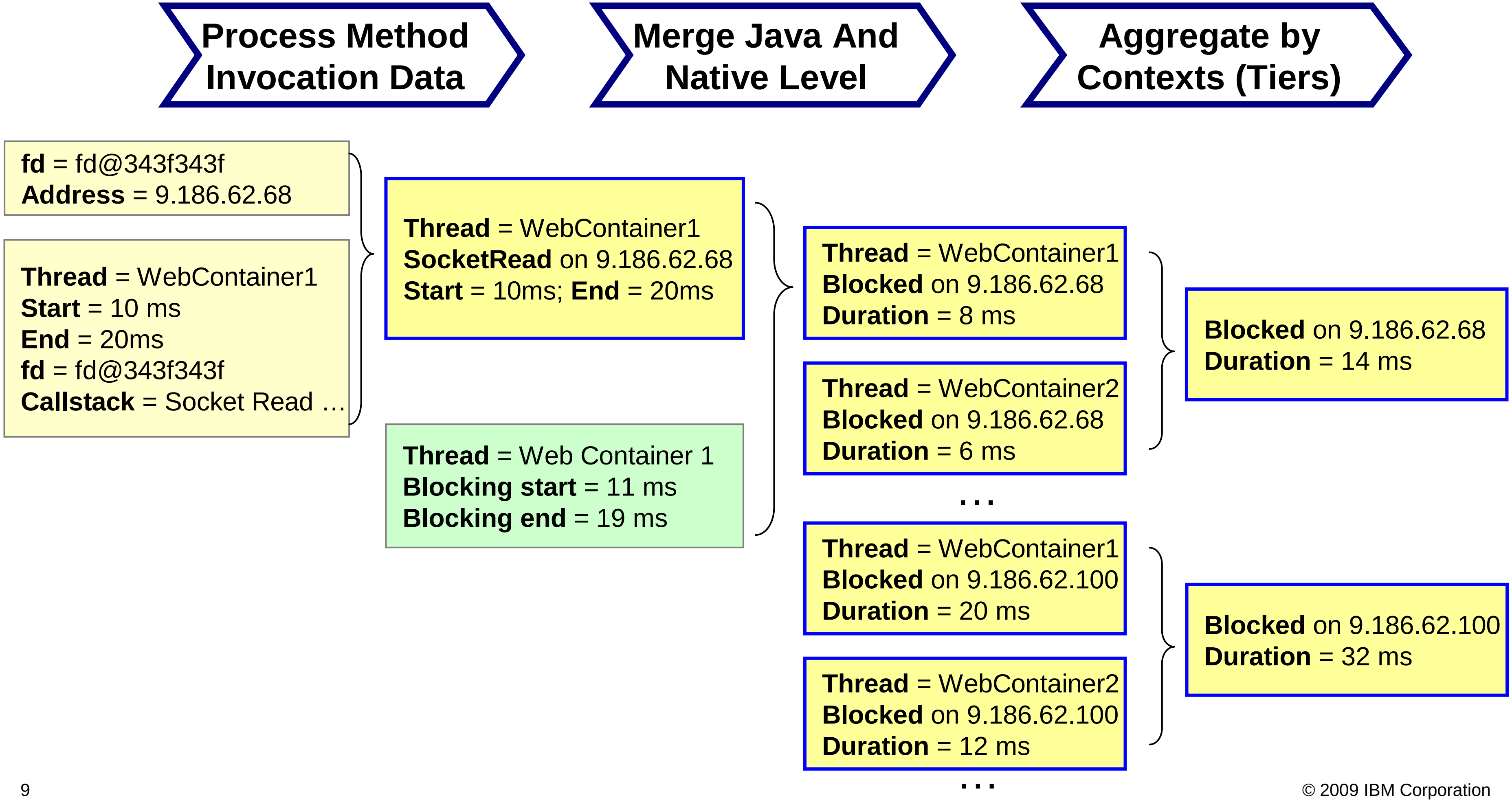


Merge the Data – Analysis Engine

Method Invocation Trace

Thread Blocking Trace

Analysis Result



Process Method Invocation Data

Merge Java And Native Level

Aggregate by Contexts (Tiers)

fd = fd@343f343f
Address = 9.186.62.68

Thread = WebContainer1
Start = 10 ms
End = 20ms
fd = fd@343f343f
Callstack = Socket Read …

Thread = WebContainer1
SocketRead on 9.186.62.68
Start = 10ms; **End** = 20ms

Thread = Web Container 1
Blocking start = 11 ms
Blocking end = 19 ms

Thread = WebContainer1
Blocked on 9.186.62.68
Duration = 8 ms

Thread = WebContainer2
Blocked on 9.186.62.68
Duration = 6 ms

…

Thread = WebContainer1
Blocked on 9.186.62.100
Duration = 20 ms

Thread = WebContainer2
Blocked on 9.186.62.100
Duration = 12 ms

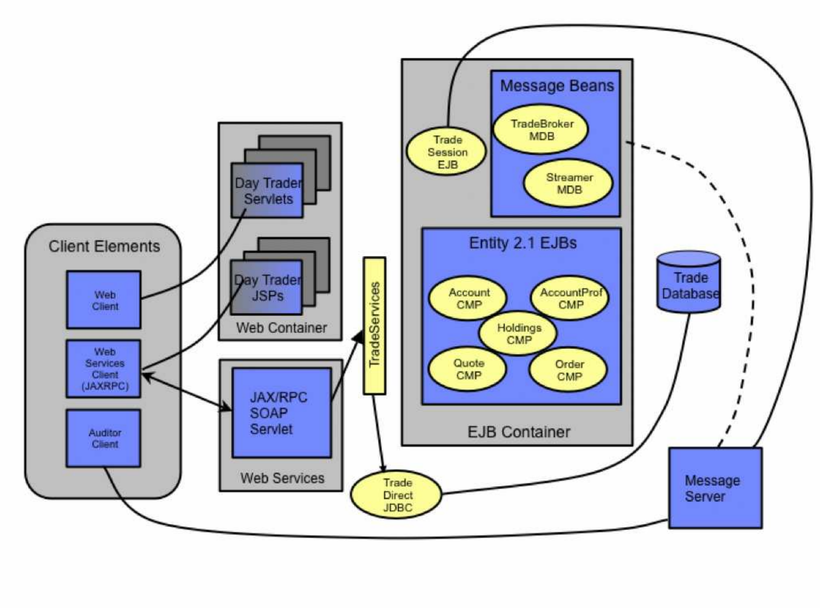
…

Blocked on 9.186.62.68
Duration = 14 ms

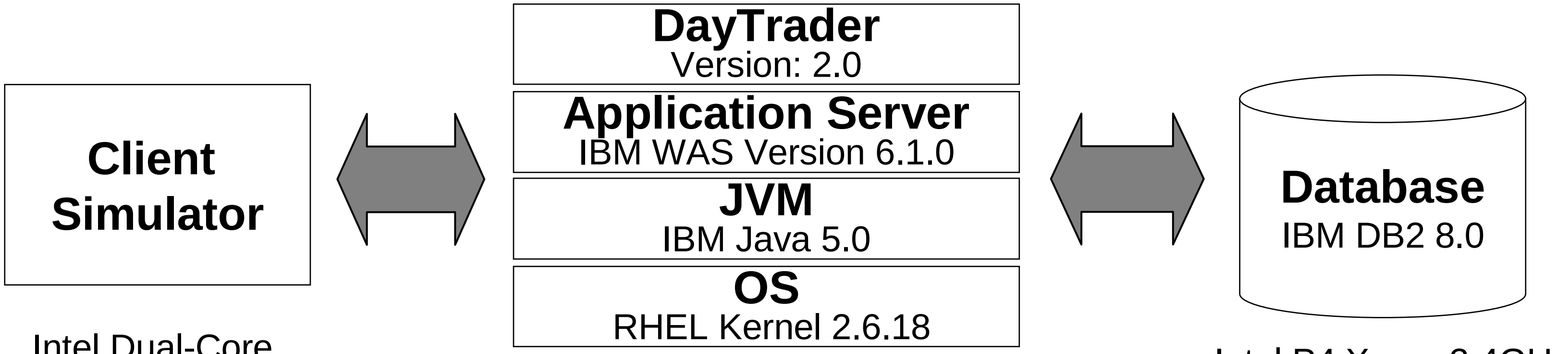
Blocked on 9.186.62.100
Duration = 32 ms

Case Study

- DayTrader
  - –Multi-tier architecture
  - –J2EE application
  - –Simulate Stock Trading



- Deployment Details in the Study



Client Simulator

Intel Dual-Core
Xeon LV 1.66GHz,
2 Processors, 4G Memory

DayTrader
Version: 2.0

Application Server
IBM WAS Version 6.1.0

JVM
IBM Java 5.0

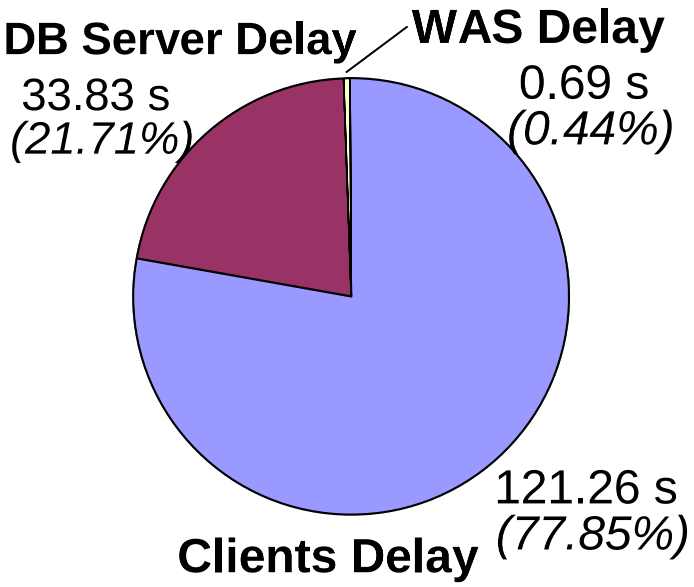
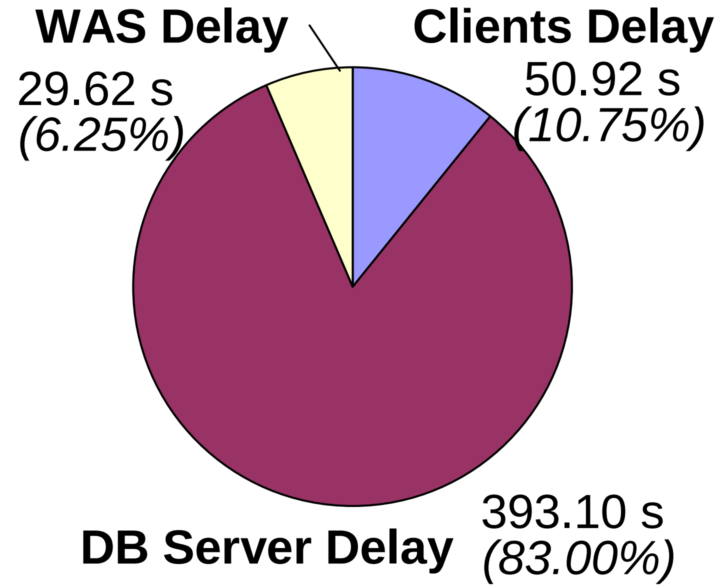
OS
RHEL Kernel 2.6.18

Intel Xeon 5345, 2.33GHz
2 Processors, 4G Memory

Database
IBM DB2 8.0

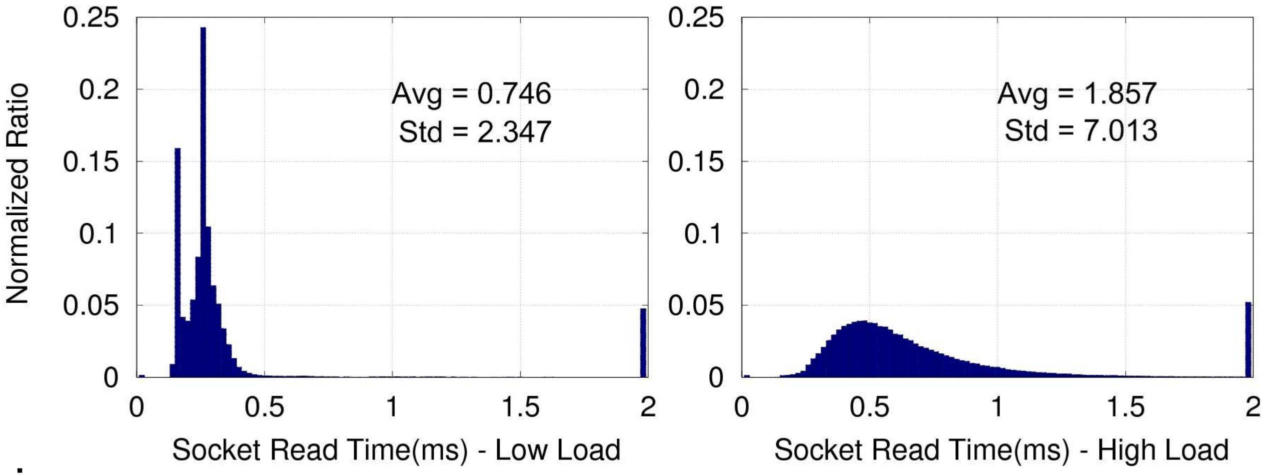
Intel P4 Xeon, 2.4GHz
1 Processor, HT Enabled
2G Memory

Low Clients Loads → High Clients Loads

Config	Low Clients Load	High Clients Load
Load	Limit client requests to 650/s	Increase client requests to 2900/s
Utilization	WAS CPU % = 6.5%	WAS CPU % = 30.9%
Cross Tier Wait Time Analysis from Middle Tier's Perspective	 Clients cause the most cross tier waiting time	 DB server causes the most cross tier waiting time

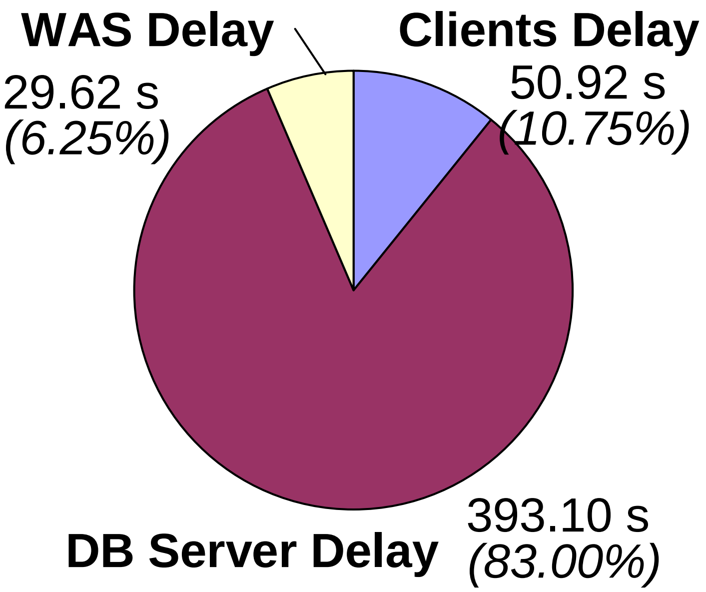
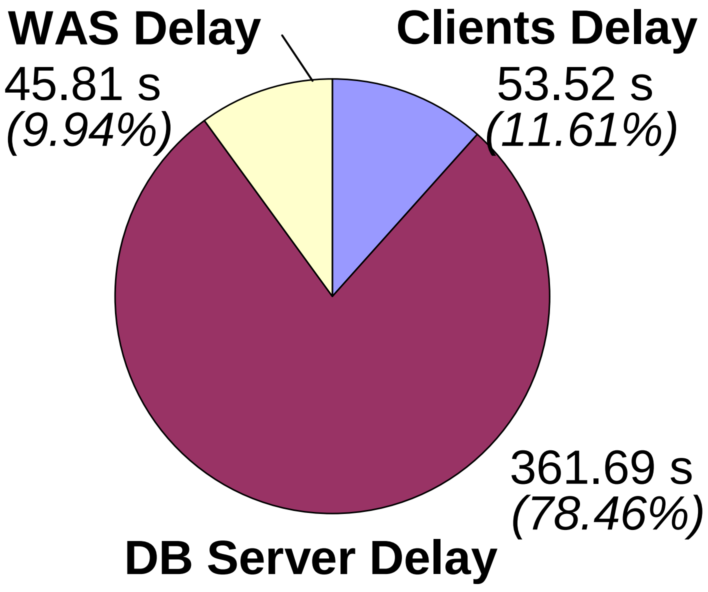
Analyzing Socket Read Time

- Cross tier delay on DB server is in SocketRead invocations
  - –Action : Study the socket read time in two loads



- Conclusion
  - –DB server’s slow response causes the low 30% utilization in WAS
- Action
  - –Upgrade DB server to 2 Xeon 5345 Processors, total 8 way.
  - –Result: Client Request Rate > 4,600/s. WAS CPU utilization = 51%

High Clients Loads → Upgrade DB Server

Config	**High Clients Load**	**Upgrade DB Server**
Load	Increase client requests to 2900/s	Reach to over 4,600/s
Utilization	WAS CPU % = 30.9%	WAS CPU % = 51%
Cross Tier Wait Time Analysis from Middle Tier's Perspective	 DB server causes the most cross tier waiting time	 Blocking time on DB server reduced.

Overhead Analysis

- Approaches Used to Reduce Overhead
  - Kernel Module
    - Only filter block threads
  - Byte Code Instrumentation
    - Only instrument selected method invocations
    - Aggressively use final and private keywords
    - Cache trace events in an array based in-memory buffer
- Resulting Overhead
  - Config: DB server uses the upgraded hardware configuration (8 way)

Tracing Rule	**Request Rate**	**Slow Down**
Base	4,699/s	0.0%
With the tool	4,169/s	11.3%

Thank you!

Q & A