Policy-based Management of a JDBC™ Connection Pool

Mick Jordan

Policy-based Management of a JDBC™ Connection Pool

Mick Jordan

SMLI TR-2006-151 February 2006

Abstract:

Managing the communication between an application server and a back-end database is essential for scalability and crucial for good performance. The standard mechanism uses a variable-sized pool of connections, but typical application servers provide very rudimentary, implementation-centric, pool control mechanisms. This requires administrators to manually translate service level specifications into the pool control mechanism, and adjust these as the load or machine configurations change. We describe the use of a resource management framework to automatically control connection pool parameters based on externally supplied policies. This simplifies the connection pool implementation while at the same time allowing a variety of policies to be applied, including policies that automatically adapt to changing circumstances.

The implementation of two distinct policies are discussed and performance measurements are reported for a contemporary synthetic application benchmark.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email address:
mick.jordan@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

Policy-based Management of a JDBC™ Connection Pool

Mick Jordan

Sun Microsystems Laboratories

1 INTRODUCTION

The bottom tier of virtually all enterprise applications is a database that provides reliable, long-term, storage of enterprise data. Client access to the database is via additional tiers of web servers and application servers that handle the large numbers of client requests and manage controlled access to the database. The mechanism for communicating with the database is commonly referred to as a *connection*. Specifically, in the Java™ 2 Platform, Enterprise Edition (J2EE™) [Sun03a], interaction is through a JDBC™ Connection [FEB03].

Database connections are expensive to create and destroy and the maximum permissible number of connections falls well short of the demands of today's internet-facing enterprise applications. This has led to the notion of *connection pooling*, where a small set of connections are multiplexed among client requests by the application server. In this case, we distinguish between *physical* connections to the database, which are pooled, and *logical* connections that are mapped transparently to the physical connections as needed. Applications only see logical connections, and are unaware of the distinction. Connection pooling has been available since the 2.0 release of the JDBC specification and is supported by most JDBC drivers.

The management of the connection pool, for example, the size and growth characteristics of the pool, is left to the server implementation and is typically application-server specific. Pools are usually characterized by simple min/max mechanisms that require manual configuration by an administrator.

In [JCK+04] we described how we modified the J2EE Reference Implementation, 1.3.1 (J2EERI) [Sun03b] to exploit the resource management facilities [CHS+03] available in the Multi-tasking Virtual Machine (MVM) [CD01]. That work provided simple controls, for example, the maximum number of physical database connections and the transaction rate. The work described here extends these controls to provide flexible and adaptive control of a JDBC connection pool.

We begin with background on the key technologies used. First, we provide an overview of the JDBC connection pool mechanism. This is followed by an introduction to *isolates*, which provide multiple applications in the Java platform, and a description of our resource management (RM) system that is built on isolates. We then describe the application of RM to control a JDBC connection pool, and illustrate several different management policies. An overview of related work and a summary conclude the paper.

2 BACKGROUND

2.1 JDBC Connection Pooling

The JDBC Connection pooling mechanism is designed to be essentially transparent to an application and also allow a variety of implementation strategies. In particular, although a JDBC driver, which creates physical connections, participates in the mechanism, it typically does not handle the pooling internally, thus allowing multiple JDBC drivers to be used with the same pooling implementation. In the J2EE context, the J2EE application server provides the connection pool implementation, which we refer to as the pool manager.

To utilize connection pooling, applications must use a `DataSource` object, which is a basic JDBC mechanism that decouples driver specifics from applications. `DataSource` objects are typically registered in the JNDI namespace and looked up by name. An application acquires a `Connection` by calling the `getConnection` method on a `DataSource` object and is unaware whether the connection is pooled or not. However, for pooling to work, the application must call the `close` method on the `Connection` object, as this initiates the mechanism that returns the connection to the pool for reuse. Closing a pooled connection does not close the underlying physical connection but merely returns it to the pool for reuse.

To minimize the dependence of the pool manager on details of the physical connection implementa-

tion, JDBC defines a PooledConnection class,[1] that is typically implemented by the JDBC driver vendor, and provides a standard protocol for the pooling manager to invoke. The PooledConnection class, which represents the physical connection, provides two important facilities: First, the ability to get a Connection instance through the getConnection method and, second, an event delivery mechanism that allows the pool manager to be informed about interesting events on the connection, in particular, when the connection is closed by an application or when an error occurs. The essential data structure that is maintained by a connection pool manager is a set of PooledConnection instances, partitioned into two subsets, free and in use.

When an application invokes the getConnection method on a DataSource object that supports connection pooling, the pool manager first looks to see if there are any unused connections in the free pool. If not, the pool manager must decide whether to increase its pool size and ask the DataSource for a new PooledConnection. This is one of the control points for resource management of the connection pool, as there are both advantages and disadvantages to increasing the number of physical connections.

In the initial release of the connection pooling API in JDBC 2.0, there was no standard way to control the pool parameters, which were, therefore, vendor specific. In the JDBC 3.0 release standard properties were defined as follows:

- **initialPoolSize**: number of physical connections to create when pool is created.
- **minPoolSize**: minimum number of connections to keep at all times.
- **maxPoolSize**: maximum number of connections allowed. Zero indicates no limit.
- **maxIdleTime**: number of seconds that an unused connection should remain in the pool before it is closed. Zero indicates no limit.
- **propertyCycle**: the frequency in seconds that the pool should check the above properties and enforce the given settings.

If adding a new pooled connection is permitted (by maxPoolSize) the new PooledConnection object is added to the free set. Otherwise the pool manager has the choice to throw an exception or to queue the request and wait for a connection to free up. This choice is not controlled by a standard property but is another common vendor-specific option [Sun04].

Given a free PooledConnection, the pool manager first removes it from the free pool and then invokes the getConnection method on it to produce a (logical) Connection object that is returned to the application. This logical Connection object is typically a proxy that keeps a reference to both the PooledConnection that produced it and the underlying physical Connection instance.

When the application invokes the close method on the logical connection object, the PooledConnection object is invoked to notify registered listeners, i.e., the pool manager, of the close event. This causes the PooledConnection instance to be placed back in the free pool for future use. Note that the physical connection is not closed and remains associated with the PooledConnection instance.

A pool manager must provide a mechanism for reducing the pool of available connections. Typical connection pool implementations periodically scan the free pool and discard connections according to the criteria defined by minPoolSize and maxIdleTime. When a connection is discarded, the physical connection is closed by invoking the close method of the PooledConnection. Note that reducing the size of the free pool is also a potential resource management control point.

2.2 Isolates

The RM system builds on another abstraction: the Application Isolation API (Isolate API), which is defined by JSR-121 [JCP01]. The main abstraction of the API is a container for concurrently executing multiple arbitrary Java programs that are referred to as isolates. An isolate can be viewed as the analog of an operating system process, and provides similar isolation and virtualization guarantees. The Isolate API includes support for creating, monitoring, and terminating isolates programmatically.

Starting a new isolated computation is similar to creating a new thread and amounts to specifying the main class and arguments for the new isolate and invoking its start method:

[1] The PooledConnection class is not visible at the application level.

```
Isolate i = new Isolate("Hello", new String[] {});
i.start();
```

The Isolate API is fully compatible with existing applications and middleware. In particular, applications written before JSR-121 may be managed by the API without modification. The Isolate API lends itself to different implementation strategies. One implementation strategy, employed by MVM [CD01], is to execute all isolated applications using a single JVM™ instance that implements the protection boundaries within a single address space.

2.3 The Resource Management API

The dependency between the RM API and the Isolate API is that the unit of management for the RM API is defined as an isolate. This choice makes accountability unambigous, as each resource in use has exactly one owner.[2]

This section contains a brief overview of the main features of the RM API [CHS+03]. It is important to note that existing applications can still run without modification, even if the classes they depend on exploit the RM system. Applications that need to control how resources are partitioned (e.g., application servers) can use the API for that purpose. Pro-active programs can use the API to learn about resource availability and consumption to improve the characteristics most important in the given case (response time, throughput, footprint, etc.) or to ward off denial of service attacks.

Key abstractions of the RM API are discussed below.

Resource Domain: A *resource domain* encapsulates a usage policy for a resource. All isolates *bound* to a given resource domain are uniformly subject to that domain's policy for the underlying resource. An isolate cannot be bound to more than one domain for the same resource, but can be bound to many domains for different resources. Thus, two isolates can share a single resource domain for, say, CPU time, but be bound to distinct domains for JDBC connections.

Constraints and Notifications: The RM API does not impose any policy on a domain; policies are explicitly defined by programs. A resource management policy for a resource controls when a computation may gain access to, or *consume*, a unit of that resource. The policy may specify *reservations* and arbitrary *consume actions* that should execute when a request to consume a given quantity of resource is made by an isolate bound to a resource domain. Consume actions may be defined to execute prior to the consume request and serve as programmable *constraints* that can influence whether or not the request is granted. Consume actions may also be defined to execute after the consume event and serve as *notifications* of resource consumption. Consume actions can cross isolate boundaries and typically do. That is, actions are usually set by a managing isolate on a resource domain that is bound to a managed isolate. Less common is an action set by an isolate to monitor its own consumption. To reduce the number of potentially expensive boundary crossings, consume actions include an associated *trigger* that executes inside the RM implementation and controls whether the callback is invoked. For example, if a policy only wishes to control a resource once it reaches a certain threshold value, it can establish a trigger that will only enable the callback when that threshold is reached. The RM API includes a pre-defined set of useful triggers, in particular one that always causes the callback to be invoked. Finally, consume actions can be specified to be invoked one-time or every-time (persistent) a client makes a consume request, and whether they execute synchronously or asynchronously with respect to the client consume request. Constraint actions are always executed synchronously and notifications are typically executed asynchronously.

Multiple consume actions can be associated with a given resource domain and, for constraints, all actions have to approve the consumption for it to be granted. The set of consume actions constitute the policy associated with the domain.

The following class defines a callback that enforces a fixed limit to the consumption of a resource. This is an example of a callback that can be applied to any resource as it is independent of the resource characteristics.

```
class LimitCallback implements ConsumeCallback.Pre {
  private long limit;
  LimitCallback(long limit) { this.limit = limit; }
  public long preConsume(ResourceDomain d, long cu, long pu) {
```

[2] In [CSH+03] we discuss problems associated with designating class loaders or threads/thread groups as principals in resource management schemes.

```
    if (pu > limit) return cu; else return pu;
  }
}
```

The preConsume method is called with the domain associated with the request, the current usage, cu, and the proposed usage, pu. Returning cu is interpreted as denying the request; returning au, where cu < au <= pu, is interpreted as approving the request, partially if au < pu.

The following code applies this callback to a resource domain d, with a limit of ten, specifying the action to be persistent, synchronous and always triggered:

```
d.setConsumeAction(PERSISTENT, SYNCHRONOUS,
                   new LimitCallback(10),
                   Triggers.newYes());
```

Defining Resources: Resources can be exposed through the API in a uniform way, regardless of whether they are actually managed by the operating system (e.g., CPU time in JVMs that rely on kernel threading), the run-time system (e.g., heap memory), core classes (e.g., file and network resources), middleware (e.g., JDBC connections), or by the application itself. Retrofitting existing resource implementations to take advantage of the RM API is relatively easy.

A particular resource is represented by an instance of a Java class that must inherit from the ResourceAttributes class, which is part of the RM API, and provides standard methods that define the resource characteristics:

- **Disposable**: A resource is disposable if it can be returned and reused at a later time. For example, JDBC Connections are disposable but CPU cycles are not.
- **Unbounded**: There is no fixed limit on the amount of resource available. CPU cycles are an example of an unbounded resource and Memory is an example of a bounded resource.
- **Reservable**: Bounded resources may be reserved by a domain for future consumption, up to a pre-determined limit.
- **Scope:** RM captures the intended scope of a given resource through an attribute that can have one of two values: local or global.

Dispensers: Dispensers model the notion of a point of manufacture for a resource. For example, a storage management system "manafactures" memory. Dispensers record the total available quantity of a resource, manage resource reservations and coordinate the consume and unconsume actions associated with resource domains.

There is a clear relationship between the scope of a resource and the number of dispensers for the resource. For global resources there is one dispenser per node. Most resources are global. However, if a resource potentially has several points of manufacture on one node, then it is deemed local and there will one dispenser per point of manufacture. Local resources typically model software artifacts that could be instantiated multiple times, e.g., a J2EE servlet container.

Exposing Resources: To make a resource manageable through the RM API, one must modify the resource implementation and insert consume and unconsume calls where appropriate. For example, the following code, invoked upon an attempt to open a new physical JDBC connection, controls allocation of new connections:

```
ResourceDomain rd =
  ResourceDomain.currentDomain(JDBC_CONNECTIONS);
long val = rd.consume(1); // request unit of the resource
if (val == 1) // go ahead if the consume request succeeded
  return new connection ...
else // consume request failed – report the failure to the caller
  fail ...
```

For bounded resources it is vital to let the RM API's accounting know that the resource has been disposed of, by planting an appropriate unconsume call in the resource's implementation. In the case of JDBC connections the code in JDBC driver's close method is augmented with:

```
rd.unconsume(1); // "return" the resource
```

Existing applications run without modifications under the control of the RM API because they do not request resources explicitly. Instead, the implementations of the resources themselves are transparently modified to interact with the RM API, so that resource consumption policies are taken into account when a resource is being used.

3 EXPERIMENTAL SETUP

This work was carried out in the context of an enhanced version of the J2EE 1.3.1 Reference Implementation (J2EERI). These enhancements are described in detail elsewhere [JDC+04, JCK+04] and we only introduce the salient aspects.

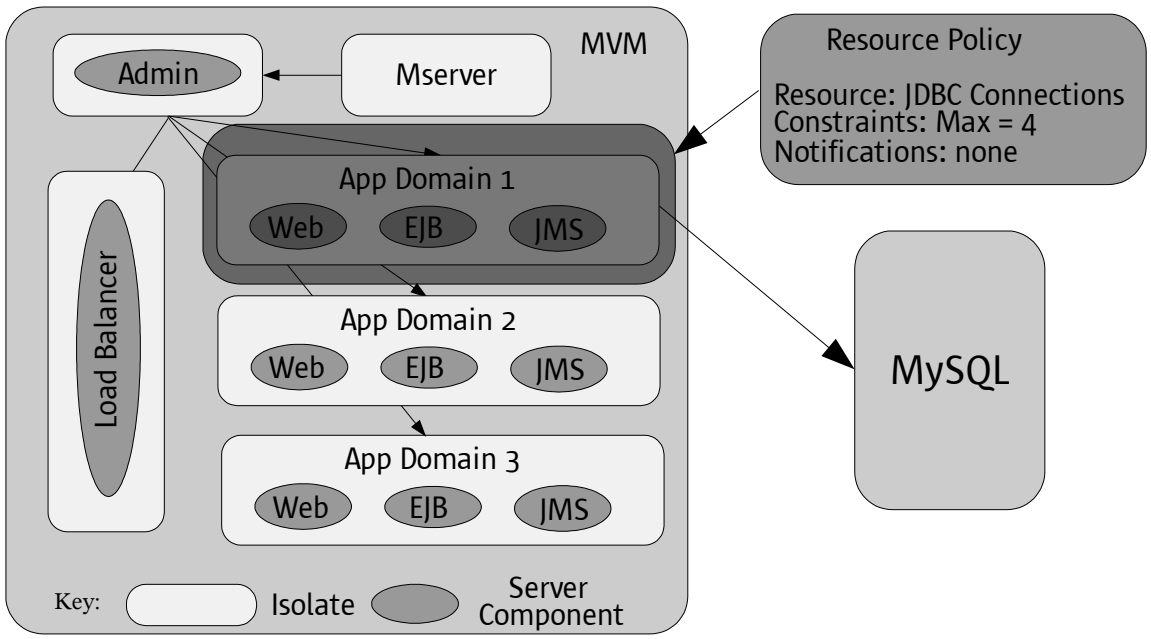

Figure 1: An example J2EERI server setup with a policy controlling JDBC connections

The enhanced J2EERI contains an administrative domain, one or more application domains and an optional load-balancer domain, each of which executes in a separate isolate. Each application domain contains all the J2EE components that one would find in the unmodified J2EERI server, e.g., servlet engine, EJB server, JDBC connection pool manager. The administrative domain handles the common aspects of J2EE application deployment. More importantly in the context of this paper, it also controls the resources that are allocated to an application domain, using the facilities of the RM API.

The RM API would allow several application domains (isolates) to be bound to a single resource domain. However, our current J2EERI implementation does not permit this and enforces the rule that every application domain has a distinct resource domain for any given resource.

A resource management policy is specified as a class that implements a standard interface defined by the administrative domain. The key method of this interface is applyPolicy which takes the resource domain to which the policy is to be applied and an array of string arguments that are policy-specific. A typical implementation of applyPolicy analyzes its arguments, if any, and then registers a consume action on the domain. Note that the consume actions are all registered by the administrative isolate, so all policies execute in that isolate.

The set of managed resources and policies is open-ended, and, following J2EERI convention, specified in property files. The application of a policy is specified as a triple <application domain, resource, policy>.

The administrative domain reads these files on startup, creates the appropriate application and resource domains and then applies the stated policies. We exploit the ability to map from strings to Java class names in order to create and access resource instances and policy instances.

4 JDBC CONNECTION RESOURCES

In our previous work [JCK+04] we had defined a resource for JDBC connections that was bounded, disposable and reservable with a granularity of one. For simplicity, this resource was implemented by modifying the MySQL JDBC driver and, as such, was really specific to MySQL. In other words the resource controls only the number of MySQL connections and not JDBC connections in general. Although it would be possible to define a resource that controls all JDBC connections, it is not clear that that would provide a useful control point. Choosing exactly which resources to manage is one of the

challenges when using a flexible system such as provided by RM.

In typical application servers, JDBC connection pools are specified on a per database-instance basis. Therefore, a resource for a pooled connection should also be database-instance specific. Given that there is always a one-to-one relationship between a pooled connection and a physical connection, it is redundant to also have a resource for the physical connection.[3] Therefore, we only consider pooled connections as a resource from here on. Figure 1 shows an example of a J2EERI server consisting of three application domains, the first of which is connected to a MySQL database server and is controlled by a resource policy that limits the maximum number of connections for that domain to four.

4.1 Pooled JDBC Connections

For simplicity and flexibility we chose to completely reimplement an RM-enabled connection pool, rather than modify the relevant portion of the J2EERI code base, which pre-dates the JDBC 3.0 specification.

Much of the code of the pool manager is independent of whether the RM system is used and can be factored into an abstract superclass. The superclass is abstract because it defines two methods, `consumePolicyCheck` and `unconsume-PolicyCheck` that control whether the pool can grow or shrink, and which must be provided by a concrete implementation. The pool size management logic is conceptually very simple. The pool manager maintains a list of available `PooledConnection` objects. When a connection is requested and the list is empty, the `consumePolicyCheck` method is invoked, If this returns true a new `PooledConnection` is added to the pool, and then immediately removed and returned to the caller. Otherwise code to handle the out of connections case is invoked. This can either throw an exception or queue the caller until a connection becomes available, depending on the pool configuration. A traditional pool manager implementation would simply encode its hard-wired logic in the methods `consumePolicyCheck` and `unconsumePolicyCheck`.

The code for the RM-enabled connection pool implementation of `consumePolicyCheck` is as follows:

```
ResourceDomain rd = currentDomain(PoolResource);
if (rd == null) return true;
else return rd.consume(1) == 1;
```

The first line acquires the resource domain that the currently executing thread[4] is bound to, if any. If the result is null the isolate containing this thread is unconstrained for this resource and therefore the increase is always permitted. Otherwise the `consume` method is invoked which, through the RM system, will consult the policy applied to this domain. If the requested quantity is granted `consumePolicyCheck` returns true, otherwise false.

Managing pool size reduction is slightly more complicated. The pool manager runs a reaper thread that wakes up periodically, determines the number of available connections, `ca`, and then invokes `unconsumePolicyCheck(ca)`, which is required to return a value, `cr<=ca`, denoting the number to reduce the pool size by. The implementation for the RM-enabled pool manager is as follows:

```
ResourceDomain rd = currentDomain(PoolResource);
if (rd == null) return ca;
else return rd.unconsume(ca);
```

Note the philosophical difference between the RM approach to controlling a pool manager and the approach implied by the property settings of the JDBC specification. In the property setting approach, the pool manager implementation is essentially exposed in the properties and the expectation is that an external controller periodically modifies these settings, which are checked periodically by the pool manager, based on the `propertyCycle` property. In the RM approach, less of the implementation is exposed and the external controller (policy object) is consulted on every resource consumption decision. Control is therefore more immediate in the RM approach and, since the pool parameters are effectively directly controlled by the policy, the pool manager implementation can be simpler.

[3] In particular, there must be at least as many physical connections available as the maximum pool size, so any policy applied to the physical connections would have to be consistent with the policy applied to the pooled connections.

[4] N.B. Since it is the isolate containing the thread that is bound to the domain, all threads in the same isolate are bound to the same policy.

Control properties can still exist in the RM approach but are applicable to the policy object not the pool manager, and are defined in terms of the policy metrics and not the pool manager implementation.

4.2 JDBC Connection Pool Policies

To test the capability of the RM system for controlling JDBC connection pools, we implemented two policies. The first is the simple bounded pool that is defined by the standard property settings. The second is a policy that limits connections such that the average latency of requests does not exceed a given value. We then discuss issues that arise with multiple application domains and possibly conflicting policies.

Simple Bounded Pool

The standard connection pool properties specify a minimum, maximum and initial pool size. We originally tested the common code of the pool manager with a subclass implementation that hard-wired this policy.

Ignoring the initial pool size property for the moment, a policy that implements this model is extremely easy to implement in the RM system. The policy code registers a consume action that is parameterized by the desired minPoolSize and maxPoolSize values for the pool. Requests to increase or decrease the pool size are checked against these limits and granted or denied appropriately. Note that a simple policy such as this is actually independent of the underlying resource since it is completely characterized by simple scalar values and the only information required from the resource implementation is carried in the generic consume callback. Therefore such a policy can be applied to any resource that fits the min/max model, for example, memory allocation. This potential for reuse is one of the benefits of the separation of policies from resource implementations.

The situation is slightly complicated by the initialPoolSize property. The benefit that this property provides is avoiding the initial connection overhead on the first client request by pre-establishing the connections.

The difficulty in implementing this policy is that there is now additional state that is private to the connection pool manager that needs to be communicated to the policy. In this particular example, the additional state is a boolean value that indicates whether we are in the startup phase or not. If so, we want to limit the number of connections to the value of initialPoolSize rather than maxPoolSize. The RM API provides no mechanism to communicate this state in the consume action.

To support an initial set of open connections, the RM-enabled pool manager must be modified to attempt to open (an unknown number of) connections on pool creation. Since it cannot know how many connections the policy is configured for, it must use an unbounded loop and detect the limit by observing the "out of connections" exception. Correspondingly, the policy implementation must assume that the first consume call is in this "initializing" state and then switch to the normal state once the initial set of connections have been granted. The RM API trigger mechanism provides a convenient and efficient way to achieve this. By associating a trigger that only fires when the usage exceeds the initial value, the callback only has to deal with maintaining the consumption between minPoolSize and maxPoolSize.[5]

Note that the policy provides adjustable properties that are very similar to the standard pool properties, e.g., minPoolSize and maxPoolSize.

Limited Latency Pool

Application server administrators who only have a bounded pool at their disposal often have to spend a lot of time tuning the bounds of the pool in order to get the performance that they want. The problem is that performance goals are typically specified in terms of throughput and latency and these measures are related in a complex way to the number of open connections and the specific application requests. Further, if the form of application queries change over time, the pool will likely need reconfiguring which, in some cases, might require a server restart.[6]

Clearly it would be better if control over the size of the connection pool were directly related to the specific throughput and latency requirements. It

[5] The policy enforces the constraint: minPoolSize <= initialPoolSize <= maxPoolSize

[6] For example, daily activities such as a catalog update, that changes the load from read-intensive to update-intensive.

was not our goal in this work to develop a particular optimal policy, as others are pursuing [MJP+04]. Our goal is to demonstrate that the RM API is capable of supporting a variety of such policies. Therefore, we chose to implement a relatively simple policy that is nevertheless representative of more complex policies.

The limited latency policy is parameterized by a single value, which is interpreted as a maximum acceptable average request latency. The policy then permits as many open connections as requested such that this maximum average latency is not exceeded on any of the open connections. To realize this requirement, the policy implementation needs access to latency measurements on the open connections that the pool is managing. This is potentially a large amount of state that must be communicated from the connection pool manager to the policy implementation. Further, since the relevant state is actually associated with the physical JDBC connection, some additional monitoring must be installed in the driver and made available.

To achieve this we modified the MySQL driver to record the count of JDBC statement executions, the minimum and maximum request latencies, and the average request latency on each connection. The information can be requested by calling the `getConnectionStats` method on the physical `Connection` object. The remaining issue is how this information is communicated to the policy implementation.

In general, only a particular policy implementation can decide what information is important and over what time interval to maintain it. Since the information is only needed on a call to `consume`, if the RM API were modified to permit passing it at the call, it would be possible for the resource implementation to accumulate the data and pass it with the call. For example, the RM-enabled pool manager could maintain an array of time-stamped `ConnectionStats` objects and send these on the call to `consume`. The policy implementation could then integrate the new data that would help it make the decision. We are currently evaluating whether to enhance the RM API to permit this style of communication.

An alternative solution is to establish a secondary communication channel between the RM-enabled pool manager and the policy implementation. Since, for flexibility, neither of these two parties should be directly aware of each other, we require a third-party to act as a broker for the channel. The administration component of the modified J2EERI already contains a Resource Manager that handles the application of RM policies and we extended this to also provide the statistics channel broker. The broker allows the registration of event listeners for resource statistics messages and listens on a well-known link for such messages from application domain isolates. A listener is registered using a key that consists of the desired message class and a set of isolates. The RM-enabled pool manager runs a thread that periodically gathers connection statistics from the open connections, using the `getConnectionStats` method, aggregates them into an array, and then sends the data on the resource link. The limited-latency policy registers a listener, which is keyed by the `ConnectionStats` class and the set of isolates that are bound to the resource domain that the policy is being applied to.

With this communication mechanism in place, the policy can maintain a model that relates the number of open connections to the measureed latency. On receiving a request to increase the pool size, it can estimate the increased latency that would occur if a new connection was opened and grant or deny the request accordingly.

5 PERFORMANCE RESULTS

We carried out two performance studies. The aim of the first study was to determine the relationship between performance and connection pool size for different hardware configurations. This study used a synthetic servlet to generate the load.

The goal of the second study was to measure the effect of the latency-control policy on an application under a varying external load. This study used the RUBiS benchmark [RUBiS], which simulates an auction website.

The experimental setup consisted of two Sun 280R servers, each with two 1015Mhz CPUs and 4GB of main memory, and a Netra-T12 server with twelve 900 Mhz CPUs and 96GB of memory. All machines were connected by a 1Gb ethernet. The software consisted of a MySQL database server and our modified J2EERI application server running under MVM. The servlet engine in this version of J2EERI is Tomcat 4.0.

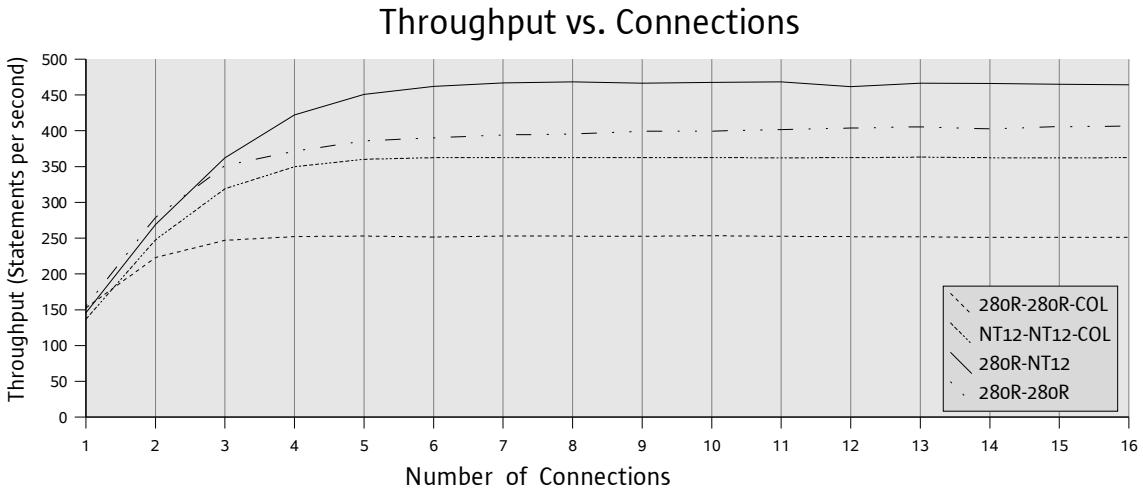

Figure 2: Throughput vs. connections for four hardware configurations

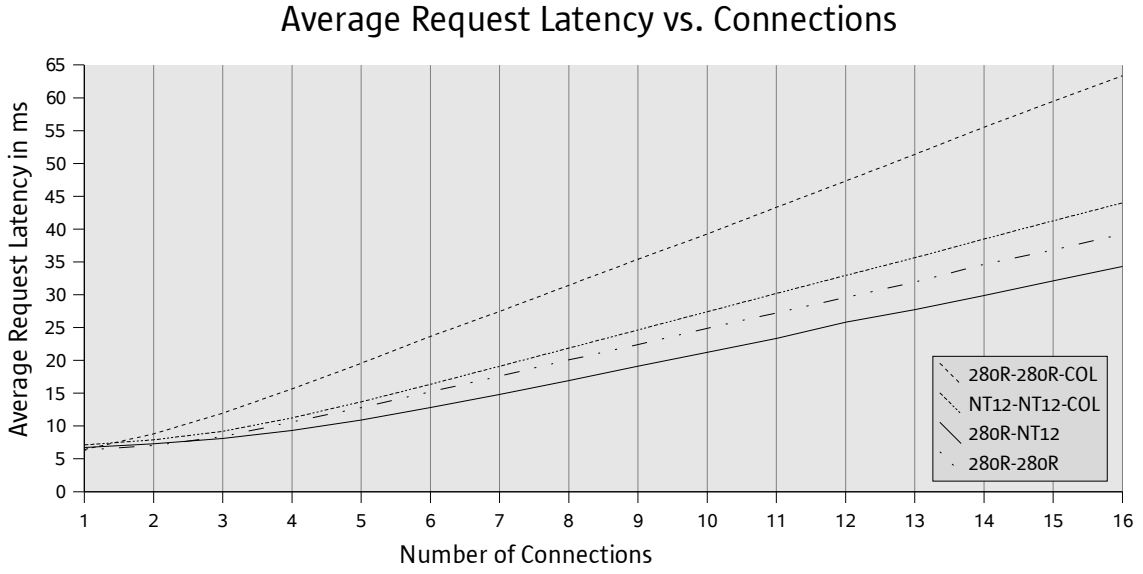

Figure 3: Average request latency for four hardware configurations

5.1 Hardware Configuration Study

To determine how performance varied with hardware configuration, we measured several different configurations of the servers, co-located on one machine and on separate machines. The load was generated by a synthetic servlet that could generate a variety of loads (using different SQL statements) and a variable number of threads each running the same query. Each thread holds exactly one, dedicated, JDBC connection and the test is run for a fixed time. We show results for four different machine configurations and a read-only query that includes a join of several tables. CPU utilization remained below 100% during all the tests.

Figure 2 shows the relationship between throughput and the number of connections, where throughput is defined as the number of SQL statements executed per second. Figure 3 shows the relationship between the average JDBC request latency and the number of connections for the same workload. Throughput rises quickly and then levels off at a maximum value, whereas latency increases linearly. Note that, although the

shape of the results is similar across the different machine configurations, the absolute values differ. This shows that, for example, that the optimal number of connections to achieve a given throughput or request latency varies with system configuration. For this query, the best throughput is achieved with the application server on a 280R and the database server on the Netra T12. For this configuration and a policy that limits average request latency to 25 ms, the maximum number of connections is eleven. However, if both servers are co-located on a 280R machine, the maximum allowable connections is six. Since the policy builds its latency model dynamically, it can handle this variance automatically.

Note that the results show the maximum throughput to be stable, with a linear increase in latency as the number of connections increases. Although not shown, we have determined that this behavior holds for up to 100 connections, which shows that MySQL scales well as connections are added. Other work [MJP+04] has shown a collapse in throughput beyond a certain connection count, in which case it is even more important to limit the number of connections to avoid operating in the thrashing zone.

5.2 The RUBiS Benchmark

In the hardware configuration study the load was fixed throughout the test, and encapsulated in the body of a single servlet request. While this allows for controlled measurements of the JDBC connection pool, it does not accurately simulate an application in an internet-facing environment, where the load varies and multiple HTTP requests are processed concurrently by the web/servlet container. The servlet programming model requires that a JDBC connection be held only for the duration of a request, with the consequence that connections are being removed from and returned to the pool at a high rate.

While still a synthetic benchmark, RUBiS provides a more accurate simulation of a real-world application. RUBiS simulates an auction trading site. The catalog of available items and the details on registered users is stored in the SQL database. The operations that can be performed by users are implemented as distinct servlets, the majority of which access the database and thus require a JDBC connection for the duration of the request. RUBiS is unusual in that it does not use the standard application server data source lookup mechanisms; instead it derives the database connection information from a properties file that is bundled in the application web archive. All RUBiS servlets inherit from a common parent class that handles connection management. We modified this class to use the RM-enabled connection pool. We also modified RUBiS to change the behavior when a connection could not be acquired from the pool. Previously this returned success at the HTTP level with an error message in the body of the returned HTML. This prevents load generators from reporting these as failed requests. Since we wanted to know when requests were failing due to connection pool limits, we modified the code to return the HTTP error as "service unavailable".

While RUBiS can be driven interactively using a web browser, it includes a tool that can generate a file of URLs, corresponding to several simulated user sessions, that can be fed to a load generator for stress testing.[7] We used the Siege load generator [Siege05] in our tests. Siege is highly configurable and can, for example, vary the number of simulated concurrent users, the length of the run, the delay between requests, and simulate internet behavior by choosing a random URL from the file for each request ("internet-mode").

To test the behavior of the system under variable load, we ran a sequence of Siege runs, steadily increasing the number of concurrent users from one to sixteen, then decreasing back to one. Each Siege run in the sequence lasted for five minutes. We set the delay between requests to zero and ran in internet-mode, which causes a continuous load to be placed on the server and the underlying database. Siege reports the number of operations submitted to the server, and the number that were successful. It also reports average response time, throughput, operation rate and effective concurrency. The latter indicates the actual number of concurrent users that could be simulated. If the server denies connection requests this number may be lower than the requested concurrency. To gather data on the JDBC

[7] By default this includes many URLs that correspond to static images displayed in the web pages. We removed these in order to focus on the database operations.

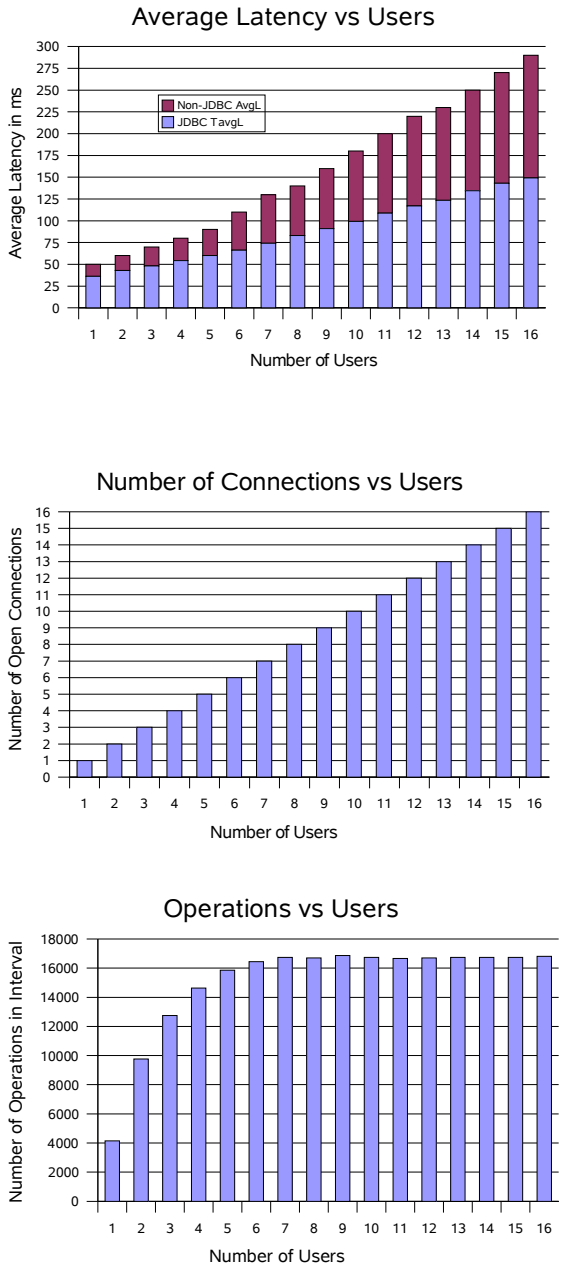

Figure 4: Performance with no controls on JDBC connection pool

connection performance, we registered a post-consume (notification) action with the resource domain for the JDBC connection pool resource. The action registered with the resource manager for the connection pool statistics and logged them at five minute intervals. Note that each RUBiS web request spawns, on average, two JDBC requests.

The first test was performed with no constraints on JDBC connection pool size, and the results are shown in Figure 4. The x-axis shows the number of concurrent users; alternatively, time moves from left to right, each slot corresponding to one five minute Siege test run with the given number of users. The three graphs show the average latency per RUBiS request, the number of RUBiS operations in each interval, and the number of

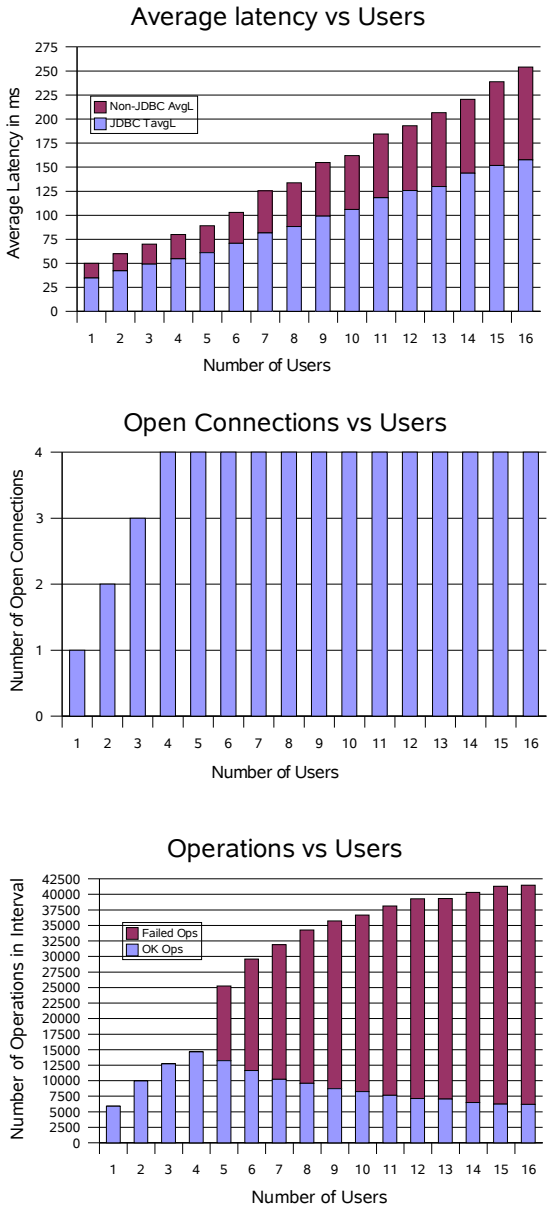

Figure 5: Performance with 30ms average latency control, throwing a SQL exception when JDBC connection is not available

connections in the pool. The latency is broken down into the JDBC and non-JDBC contributions. The JDBC contribution is the time taken to execute the JDBC statement, that is, after obtaining a connection from the pool. The non-JDBC connection contribution is everything else, which includes network latency, time spent in the web container and time acquiring the JDBC connection. We only show the increasing user phase of the test as the decreasing user results were symmetric. These results are similar to those from the hardware configuration experiment and show that each simulated user requires a corresponding JDBC connection, and that JDBC request latency varies linearly from 36ms to 150ms as the number of connections varies from one to sixteen. Note also that the non-JDBC latency is increasing at a faster rate, suggesting that the HTTP Connection system of the web container is not managing the load effectively.

The second test applies the average-latency-control policy to the JDBC connection pool resource with a limit of 30ms. That is, for each web request, the total JDBC latency contribution should not exceed 60ms. The results from this run are shown in Figure 5.

Note that although the connection pool size is capped at four by the policy, the average latency does not meet the policy limit across the range. While the cap of four connections initially meets the policy limit, latency continues to increase as the number of simulated users increases. This behavior is due to there being no controls on the number of web connections and the JDBC connection pool being configured to throw an exception when a connection is not available. Therefore, the number of active threads continues to increase in line with the number of users and each thread performs a significant amount of work before learning that the JDBC connection resource is unavailable. In other words, the system starts to thrash, spending ever increasing amounts of time on ultimately wasted processing. The threads that do get JDBC connections therefore are scheduled less frequently during their execution which explains the increased JDBC request latency.

There are two approaches to address this problem:

- Queue (suspend) threads until a JDBC connection becomes available.

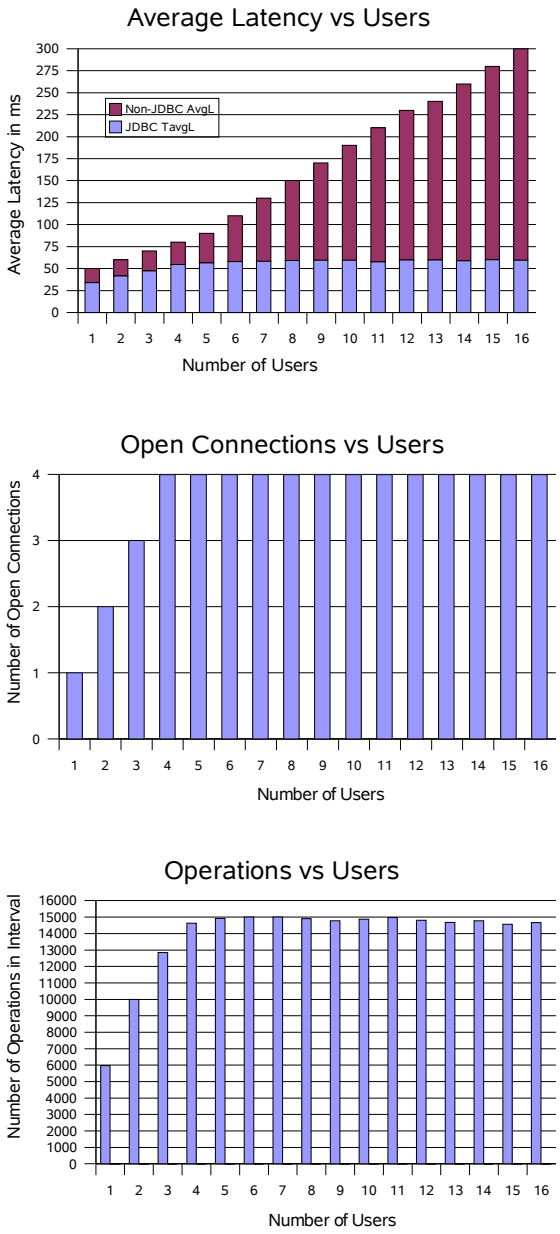

Figure 6: Performance with 30ms average latency control, queueing caller when JDBC connection not available

- Reject web connections that would result in a failure to acquire a JDBC connection.

Note that the first option will increase the average latency experienced by the end-users, but service all users eventually, whereas the second option will provide guaranteed average latency but reject some percentage of the user requests.

The first approach is easy to achieve as the ability to queue callers is an option on the JDBC connection pool. Figure 6 shows the results. Note that all requests now succeed but, while the JDBC latency stays within the specified limit, the overall request latency increases considerably, due to the queueing.

Rejecting web connections requires the web server to be modified to allow RM-based control of the number of HTTP connections or servlets. The latter would be optimal since it is the servlets that perform the database access whereas, in general, an HTTP connection could be accessing a static web page.[8] Since the web server had existing controls only on HTTP connections we chose to manage these. By default, the web server maintains a pool of threads for processing HTTP requests, with a fixed upper bound set at server startup time. It was easy to modify the code that allocates new HTTP processors to invoke the RM framework using a newly defined resource type, HttpProcessor. The remaining issue is the policy to attach to the associated resource domain to avoid thrashing. The first test (Figure 4) showed that one HTTP connection could fully utilize one JDBC connection. Therefore, we could simply attach an instance of the average-latency control policy. However, this would be somewhat wasteful as both the HttpProcessor and the JDBC Connection Pool resource domains would be building identical models. Evidently there is a relationship between the consumption of these resources in that the successful allocation of a web connection will result in a JDBC connection request that, if the pool is full, will attempt to increase the pool size by allocating a new physical connection. This relationship can be captured by defining a new policy that delegates consume requests to the policy associated with the dependent resource. While this requires additional infrastructure, it avoids duplication and is more flexible. For example, if it was determined that each HTTP connection could only utilize 50% of a JDBC connection, the policy attached to the web connection could delegate on every other callback. Note, however, that determining this ratio dynamically would require closer coupling between the policies for the two resources.

[8] Recall, however, that all static page URLs were removed from the set used in the tests.

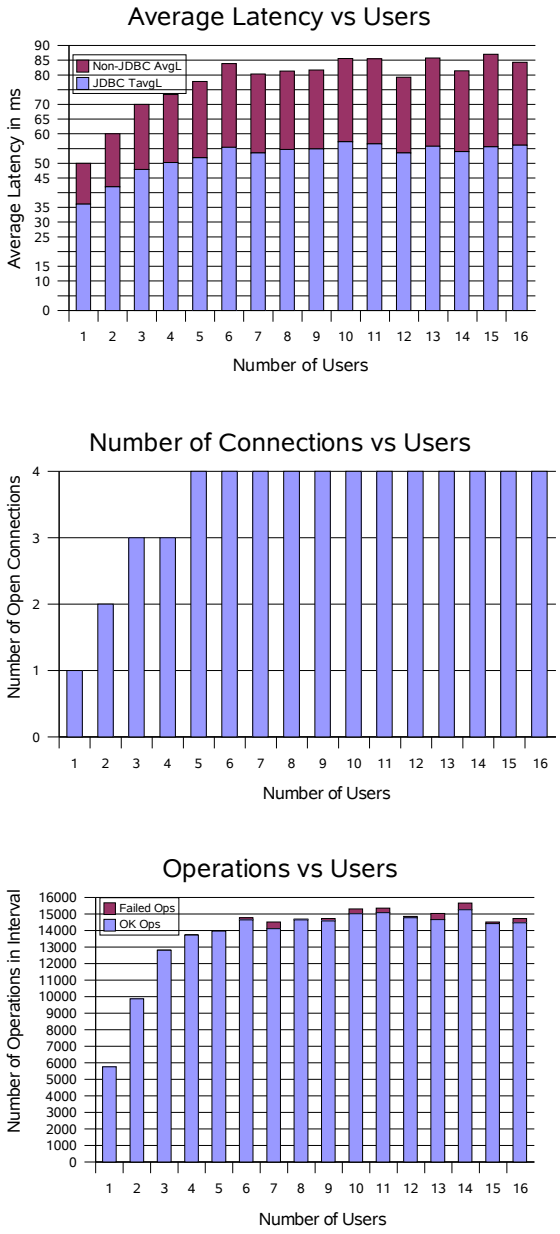

Figure 7: Performance with 30ms average latency control, rejecting HTTP connection if JDBC latency would be exceeded

To support policy delegation we extended the policy interface with an additional method that returns a delegation callback or null if delegation is not supported. Providing a separate callback allows the delegated-to policy object to distinguish delegated and non-delegated consume requests. The delegation policy takes the name of the delegated-to policy as an argument, looks it up

in the set of policies associated with the application domain, and then registers a consume callback that invokes the delegation callback of the target policy. The results from this configuration are shown in figure 7. [9]

Unlike the previous tests, these results are averaged over several runs to counteract apparent non-deterministic behavior of the HTTP processor allocation code. Occasionally the code would try to allocate several additional processors over a short interval, despite the offered load from siege apparently remaining steady. These requests were typically rejected by the latency control policy, but then we would see JDBC connections being closed and removed from the connection pool, as if some of the HTTP processor threads had ceased activity. This caused temporary drops in throughput and reduced latencies. We do not understand the cause of this behavior but averaged over a sufficient number of runs these effects are smoothed out.

Comparing the average latency graph of figure 7 against figure 6, it is seen that not only is the JDBC latency held to the policy limit, but the total latency as seen by the end-user (siege) also becomes independent of the offered load as the load increases. Compared to figure 5, there are very few failed requests due to not being able to get a JDBC connection. The trade-off is that many users are unable to get an HTTP connection at all, but those that do get a connection are serviced in a short space of time. Throughput is high across the range meaning that the server is maximizing useful work.

6 MULTIPLE CONNECTION POOLS

The previous discussion assumes that there is only one connection pool for a particular database instance. Imagine, however, that we have two application domains, both of which have a connection pool to the same database instance. One motivation for multiple application domains, and hence pools, comes from wanting to apply different policies to different clients of the system. For example, "gold" level clients are, say, guaranteed faster response times and/or greater throughput than "bronze" level clients. Since the database interaction is a significant part of any client interaction, it is tempting to think that we could achieve part of this differentiation by applying different policies to the associated connection pools.

Unfortunately, in the case of multiple connection pools to the same database instance, operations on one pool can affect the others, since the database cannot distinguish the connections from the different pools. This can therefore render the applied policies ineffective. For example, the simple latency model is based on being able to accurately predict the increment in latency that results from adding a new connection. If multiple pools interleave their requests to increase pool size, then, depending on the arrival times of the requests and the connection statistics, the latency predictions may be underestimates, which will result in violating the latency limits.

The key observation is that the underlying database instance is a resource that is global to all application domains and this must be reflected in any higher-level resources that depend on it. Although such a resource may be partitioned among several application domains, it is incorrect to treat the domains as independent. In particular, since request latency is proportional to the total number of active connections, it should not be possible to have multiple application domains with different average request latency limits, since all connections across all domains will necessarily exhibit a similar latency. Although throughput can be controlled on a per-application domain basis by using request-rate limiting policies,[10] it is still the case that multiple pools must be coordinated since the maximum throughput per connection is inversely proportional to the total number of connections.

Resources that are global will always be managed by a single dispenser. However, the control that is provided by the dispenser is limited to managing reservations and the total available quantity of bounded resources. This degree of control is inadequate for our purposes. However, multiple consume actions may be associated with a single resource domain and the dispenser ensures that all actions approve the allocation before granting the request. While this simple mechanism always enforces the most restrictive policy, it is adequate

[9] Note that the scale of the latency axis is much reduced compared to Figure 6.

[10] Thereby increasing the overall latency of a request by introducing extra delays.

for our purposes and allows a latency control policy to override any other policies that are in effect.[11]

We note in passing that the RM reservation mechanism does provide a simple, built-in, way to partition resources between domains through the use of reservations. Subject to existing usage, reservations can be changed, thus providing a crude form of inter-domain control. However, this is really abusing the reservation mechanism, which is intended to ensure that, at some future time, a minimum amount of resource can be allocated, while avoiding the overhead of allocating the resource in the interim. One valid use of a reservation in the connection pool context would be to reserve the number of connections specified by minPoolSize, thereby guaranteeing that all pools can at least operate at their minimum levels.

Managing Multiple Connection Pools

By default the J2EERI resource manager assumes that policies are application domain specific and independent. Therefore, it invokes the applyPolicy method for each policy/resource domain pair and this typically creates a new policy object which registers a new consume action with the given resource domain.

Policies that require inter-domain coordination typically need to control other domains beyond the domain they are specified for. Therefore, they need access to all the resource domains for the given resource. This information is not available directly from the RM API, but the J2EERI Resource Manager can provide it, since it is responsible for creating all the resource domains.

With this information in hand, the limited latency policy can monitor the average latency across all connection pools in all domains by registering to receive the connection statistics for all domains. Secondly, it can apply the latency control policy to all domains. For example, consider the case where one domain is operating under a minmax policy and another under a latency control policy. Unless the latency control policy is also applied to the minmax domain, connection allocations in the minmax domain will perturb the latency domain causing policy violations. However, if the latency policy is applied to both domains, the latency policy will override the minmax policy and hold both domains to a sum total of connections that honors the latency policy limit.

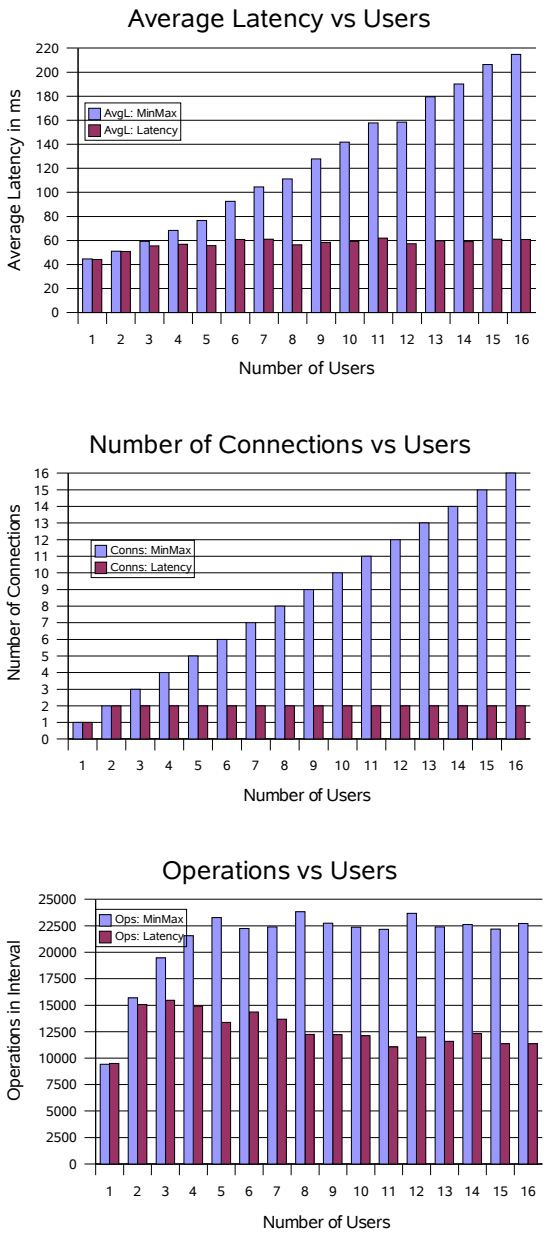

Figure 8: Performance of two domains with latency control solely on domain 2.

[11] Earlier versions of RM supported a pluggable algorithm to combine the results of multiple consume calls. We now believe this is best handled at the policy level.

To test the multiple connection pool case we ran the application server with two application domains, each driven by a separate instance of siege. In the first test, one domain operated with a minmax policy and the other with a latency control policy, i.e., we did not attempt to impose the latency control globally.

Figure 8 shows the results for the two domain test described above, with latency control only applied to domain 2. For clarity of comparison between the two domains we only show the JDBC latency contribution.

Surprisingly, these results were at variance with the hypothesis that connection latency would be similar across all connections in both pools. While there was an increase in average latency at a given pool size over the single domain tests, the effect was much smaller than expected. In particular, while the latency-controlled pool was limited to only two connections instead of four, the latency on these connections remained essentially flat as the minmax pool grew to sixteen connections, with the latter exhibiting the expected linear latency growth.

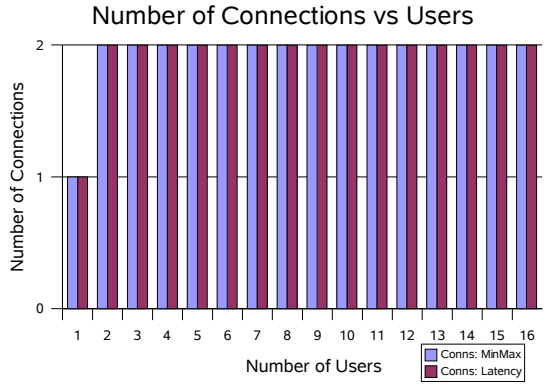

Figure 9: Latency policy override on connections

While these results demonstrate that differentiated service can be achieved with RM, they are hard to explain given that the database instance is shared and is unaware of the domain structure in the application server. We can also rule out thread scheduling effects as this is also managed by code that is unaware of the existence of isolates. That is, any latency caused by the increasing number of threads should affect both domains equally.

Since the latency measurement includes the time spent in the code of the JDBC driver, we initially conjectured that the observed behavior could be explained by attributing the major latency component to the driver. Since the driver, and its associated data structures, e.g., queues, is replicated in the two domains, the length of such queues would be proportional to the number of connections and therefore could explain the latency variation. To test this hypothesis, we ran a simple experiment that varied the number of application domains from one to sixteen, and limited each domain to exactly one JDBC connection. Unfortunately, the results did not support the hypothesis as the latency showed the same linear increase as occurs when the connections are all allocated in a single domain. Currently we are unable to explain the unexpected behavior.

That aside, Figure 9 shows the effect of applying the latency control policy to both domains, that is, allowing the latency control policy to override the minmax policy. We only show the effect on JDBC connections; the other results are similar to Figure 8.

7 RELATED WORK

We have found very little published work on automated management of JDBC connection pools.

Middle-R [MJP+04] is a cluster-based database replication tool that includes an adaptive connection pool with a hard-wired, but highly adaptive, algorithm. The algorithm focuses on maximizing throughput, rather than request latency, in the face of changing loads. Their algorithm is embedded in the connection pool implementation and achieves its effect by varying the number of open physical connections in a similar way to our work. What distinguishes our work from Middle-R is the division of the connection pool management algorithm into a policy-independent part that resides in the connection pool implementation and one of several policy algorithms that can be applied externally. The Middle-R work also does not address the issues with multiple connection pools.

8 CONCLUSIONS

We have shown how to structure the implementation of a JDBC connection pool manager so that the decision to increase or decrease the number of physical database connections can be delegated to a separate policy-agent. We described an agent the operates in the context of the Resource Management framework available with the Multi-tasking Virtual Machine. Two policies were presented, one that implements the traditional minmax style connection pool and one that attempts to limit the request latency on connections in the pool. The latter policy required a mechanism for communicating additional statistics from the resource implementation to the policy-agent, that was implemented using events routed through the application server resource manager. In the future, it may be beneficial to support this communication directly within the RM API.

Detailed measurements of the effect of the policies on the RUBiS benchmark application were reported. The results showed that request latency could be controlled. However, failure to control related resources, in particular, the number of concurrent web connections could adversely impact the effectiveness of the policies. The solution to delegate control of the web connection resource to the JDBC connection pool resource was demonstrated to be simple and effective.

Finally, the case of multiple connection pools with possibly different policies was considered. While the conjecture that the JDBC request latency was primarily due to the external database suggested that independent control would not be possible, the results surprisingly showed otherwise. We expect to revisit this issue in future work. Nevertheless, it was demonstrated that the RM framework could enforce a global policy by supporting multiple policies on a single domain and by deferring to the most restrictive policy.

9 ACKNOWLEDGEMENTS

We would like to thank Laurent Daynes, Darpan Dinker and Jeanie Treichel for their careful reviews and helpful comments.

10 REFERENCES

[CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution.* 17th ACM OOPSLA'01, Tampa, Fl, October 2001.

[CHS+03] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. *A Resource Management Interface for the Java™ Platform.* Sun Microsystems Laboratories Technical Report 2003-124. June 2003.

[FEB03] Fisher, M., Ellis, J., and Bruce J. *JDBC API Tutorial and Reference, Third Edition*. Addison-Wesley, 2003.

[JCP01] Java Community Process. JSR 121: Application Isolation API. http://jcp.org/jsr/detail/121.jsp.

[JCK+04] Jordan, M., Czajkowski, G., Kouklinski, K., and Skinner, G. *Extending a J2EE Server with Dynamic and Flexible Resource Management.* ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, ON, October 2004.

[JDC+04] Jordan, M., Daynes, L., Czajkowski, G., Jarzab, M., Bryce. C.: Scaling J2EE™ Application Servers with the Multi-Tasking Virtual Machine. Sun Microsystems TR 2004-135 (2004)

[MJP+04] Milan-Franco, J., Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B. Adaptive Middleware for Data Replication, Lecture Notes in Computer Science, Volume 3231, December 2004, Pages 175 - 194

[RUBiS] Rice University Bidding System. http://rubis.objectweb.org

[Siege05] http://www.joedog.org/siege/

[Sun03a] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/index.jsp

[Sun03b] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE) - Version 1.3.1 Release*. http://java.sun.com/j2ee/sdk_1.3/

[Sun04] Sun Microsystems, Inc. *Sun Java System Application Server Enterprise Edition 8.1 2005Q1 Performance Tuning Guide.* http://docs.sun.com/source/819-0084/pt_tuningas.html#wp116227

ABOUT THE AUTHOR

Mick Jordan is a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include programming languages, programming environments, persistent object systems and systems software. He has a Ph.D in Computer Science from the University of Cambridge, UK. He was a member of the team that designed and implemented the Modula-3 programming language. Currently he is working on the JOE project which is investigating and prototyping technologies to support Java Operating Environments.