The Definitive Guide to Terracotta: Cluster the JVM™ for Spring, Hibernate, and POJO Scalability

ISBN-13 (pbk): 978-1-59059-986-0

ISBN-10 (pbk): 1-59059-986-1

ISBN-13 (electronic): 978-1-4302-0640-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Jeff Genender
Development Editor: Matthew Moodie
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Tracy Brown Collins
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Production Editors: Jill Ellis, Laura Cheu
Compositor: Gina Rexrode
Proofreader: Linda Seifert
Indexer: Toma Mulligan
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Theory and Foundation Forming a Common Understanding

Terracotta is Java infrastructure software that allows you to scale your application for use on as many computers as needed, without expensive custom code or databases. But Terracotta is more than this twenty-six-word definition. If you have picked up this book, you are looking for a deeper understanding of the technology and where to use it. To achieve that understanding, we must first break down the jargon surrounding scaling, clustering, and application architecture. In Chapter 1, we will spend some time defining terms and then explain the value Terracotta provides to applications. This will help us to build a shared understanding of the target audience of the technology.

Applications always have a developer and operator, although sometimes one individual may serve in both roles. The application developer works to deliver business logic that functions to specifications and is responsible for all the application code. The operator manages the production application and works with the developer to make sure the application is scalable enough to handle its potential production workload. The two must work together if the application is going to be successful, meaning that it is low cost to run and stable. Terracotta was designed with both the developer and operator in mind.

Terracotta was designed to free the application developer from the constraints of enterprise development frameworks, like Hibernate, Spring, and Enterprise JavaBeans (EJB). Whether or not applications use these frameworks, close reliance on the database leads to a lack of freedom of design. While Hibernate and Spring strive to help the developer by greatly simplifying application code, many applications fall short of the promise of freedom because they are never truly decoupled from the database. Terracotta uniquely provides a complete solution for decoupling from the database and scaling while simultaneously writing pure plain old Java objects (POJOs).

Terracotta was also designed to deliver the most stable operating platform for production Java applications, period. The system can be managed in a manner similar to databases—if you can operate a database, you can operate Terracotta. And data that Terracotta is managing on behalf of an application cannot be lost. Unlike the database however, Terracotta provides solutions for developing linearly scalable applications.

While most clustering vendors focus on application scalability, Terracotta manages to deliver scalability in conjunction with the highest levels of availability, and it does all this without a database. The system also provides unprecedented visibility across the application stack in a single vendor solution. Specifically, we can now monitor applications at the operating system level, the JVM level, and the application level through a single graphical dashboard or via Java Management Extensions (JMX). This means we can monitor not just application or application server performance but also hardware performance from a central location.

The developer and operator need a simple way to scale production applications. To date, that way has been the database, but it tends to work only because a database has a large array of management tools for production operation and because its development model (SQL queries, stored procedures, and object-relational mapping) is widely understood. The database is not really what developers would ask for had we the choice however, and Terracotta's founders have seen another option. In fact, Terracotta's dual focus on developmental and operational needs did not come about by chance.

Terracotta was born in the fires of a large-scale production Java application. While the business was growing by 100 percent year on year (which may not sound all that large until we explain that revenue was in billions of dollars), the platform team held itself to a standard of purchasing no new hardware after the first year of operation. The team came very close to meeting that goal for four years in a row. Terracotta takes lessons learned at that top-ten e-commerce website and significantly advances the state of the art of scalability by creating a general purpose extension to Java that any application can use to reap the benefits of simple development with simultaneously scalable and highly available production operation.

Clustering was once thought of as complex and highly unstable due to sensitivity to individual server failure, but with the advent of Terracotta, clustering is an idea whose time has finally come. Through the pages of this book, you will learn Terracotta's unique definition of clustering. The first four chapters focus on concepts and principles and the next few show how to apply Terracotta to common use cases such as clustered caching with Hibernate, clustering in Spring, HTTP session replication, and more.

Once these chapters have whet your appetite, this book will transition into covering both detailed techniques for building powerful abstractions using Terracotta and how to tune for any and all Terracotta usage. Now, let's dive into a more detailed definition of the technology.

Definition of the Terracotta Framework

As we've already stated, instead of filling your head with jargon by defining Terracotta using otherwise unclear terms, we will give a short definition and an explanation of the benefits of the approach. We'll break down each term in that definition into plain English, so that we have a strong basis from which to proceed to use cases.

Terracotta is a transparent clustering service for Java applications. Terracotta can also be referred to as JVM-level clustering. Terracotta's JVM-level clustering technology aids applications by providing a simple, scalable, highly available world in which to run.

To understand this definition, we need to define "transparent clustering service," as well as the operational benefits, including "simplicity," "scalability," and "high availability." We can then discuss the applications for this technology.

- *Transparency*: To users, transparency means that an application written to support Terracotta will still function as implemented with zero changes when Terracotta is not installed. This does not mean Terracotta works for everything or that Terracotta is invisible and does not require any changes to application code. Transparency for the Terracotta user denotes a freedom and purity of design that is valuable, because the application stays yours, the way you wanted to build it in the first place.

- *Clustering*: Clustering has had many definitions, and as a result, Terracotta struggled for some time with calling itself a clustering service. A clustering service is unique in that, while clustering refers to servers talking to each other over a network, a clustering service refers to technology that allows you to take an application written without any clustering logic or libraries and spread it across servers by clustering in the JVM, below the application. This moves clustering from an architecture concept to a service on which applications can rely when running in a production environment.

- *Simplicity*: Complexity and its antithesis simplicity refer to the changes that developers have to make along the road to building scalable applications. For example, to make an application scalable, objects may have to implement the `Serializable` interface from the Java language specifications in order to share those objects across servers.

- *Scalability*: Scalability can be confusing. This is a time when a background in statistics helps, because developers often confuse many of our performance-measuring terms, such as mean versus median and scalability versus performance. Scalability is important if you want to save money by starting small and growing only when demand requires. Scalability is the result of low latency and high throughput. An application can be very quick to respond but might only handle one request at a time (low latency and low throughput). Alternatively, an application can be very slow to respond but handle thousands of concurrent requests (high latency and high throughput), and the database is a great example of this latter type of application. Unlike the traditional relational database, Terracotta helps optimize both latency and throughput to create a truly scalable application.

- *Availability*: Availability means that every piece of shared data must get written to disk. If the datacenter in which the application cluster is running on top of Terracotta loses power, nothing will be lost when the power is restored. No individual process is critical, and the application can pick up where it left off.

With this common terminology, we will next look at analogies to Terracotta that already exist in the datacenter environment.

Network Attached Storage Similarities to Terracotta

Terracotta is a transparent clustering service, which can be used to deliver many different things such as session replication, distributed caching, and partitioning (otherwise known as grid computing). Transparency, as we have defined it, allows applications to depend on Terracotta without embedding Terracotta. There are several transparent services on which applications rely without embedding those services inside our application source code.

File storage is an example of a transparent service. Files can be stored locally on an individual server. File storage devices can change from tape to CD to hard disk without changing our application source code. What's more, file system performance can be tuned without changing our application. Files represent a popular mechanism for storing application data because they provide transparency of storage location, storage format, and performance optimization.

Terracotta brings the same capabilities to Java application memory. Thus, while networked file storage services are not a perfect parallel to Terracotta, file I/O is worth exploring a bit for these similarities. Their similarities should help you understand the value of transparent services before we jump into Terracotta and JVM-level clustering.

The file API can be summarized as having `open()`, `seek()`, `read()`, and `write()` methods. Underneath these methods, many different behaviors can be bolted on, each with specific strengths. A file can be stored on various types of file systems, allowing us to optimize performance for certain applications completely outside of source code.

Some file systems in use today include variants of the Journaled File System (JFS) architecture, the High Performance File System (HPFS), the Berkeley Fast File System (FFS), the MS-DOS File System, and Solaris's Zettabyte File System (ZFS). Each of these technologies has a specific performance character and sweet spot. For example, journaling is very good for production systems, because changes to the disk can be rolled back to repair file corruption that a bug in an application might have caused in the recent past. JFS is, therefore, well suited to highly available data. The Berkeley FFS is very good for large-scale random access usage underneath a multiuser server, where many tasks are writing to many files of different sizes for varying reasons at random times. Here, explicit fragmentation leads to predictable latency and throughput as a result of the random and even distribution of all data. Taking the opposite approach to Berkeley's file system, the DOS file system was tuned for sequential access underneath word processing applications in dedicated user scenarios. Microsoft's original file system approach worked well, because individual files were contiguous and in one place on the hard disk.

A Layer of Abstraction

It is quite interesting to see that these file systems each handle certain use cases very well and others not as well (we have all had to wait while the Microsoft Windows defragmenter rearranged our hard drives overnight for faster access). Nonetheless, when Microsoft changed its operating system architecture and adopted some of the HPFS or Berkeley techniques, many applications were none the wiser. No programs had to be rewritten if they used only the standard file APIs. Of course, any application that made assumptions about the performance characteristics and capabilities of the DOS file system had to change when the underlying file system changed.

Terracotta users find a similar abstraction with clustering that applications find with file systems. Terracotta users do not make assumptions about how objects move across the network, nor do the users write code explicitly to partition data among Java processes, as an example. Just like the operating system can switch file systems in production without changing an application's source code, Terracotta can run different optimizations (about which you will learn in subsequent chapters) without requiring the application to change.

Similarly, we can change the file system used by a production server to write to disk without going back to development and rewriting the application, and we can move the task of storing files outside any one server's scope. If a file is on a remote file server, the application does not care. Operations teams decide which file system to use. They may choose a file system technology because it makes the application run faster or choose a remote file server because it makes the application easier to manage or because the application can use the Network File System (NFS) protocol as a central broker for sharing information across processes.

Adding Networked File Storage

Exploring further, Network Appliance was founded in April 1992 to exploit the notion of network file storage for building large application clusters. At the time, Sun was dominating the Unix market as the premier platform on which to manually construct networked file servers using the NFS protocol that shipped inside most Unix variants. Novell was dominant in the PC market, providing similar networked file and logon services for MS-DOS and Windows users.

Network Appliance realized that setup, maintenance, and ongoing operation of networked storage was too expensive and decided to produce a hardware device that honored the NFS specification but required no setup other than plugging it in. The value of networked file storage is that the storage can be managed independently of the application server. The storage can be made highly available through well known backup and recovery techniques, such as backing up to expensive centralized tape robots or replicating data on a regular schedule to a disaster recovery site. The storage server can also be scaled to meet the I/O needs of the application without simultaneously buying bigger CPUs and more expensive application server hardware.

Parallels to Terracotta

Terracotta offers the operator benefits similar to networked file storage. Specifically, objects are stored remotely in Terracotta even though the objects are cached inside the JVM for fast application access. This means that the operator can focus on hardening the Terracotta server to deliver high availability, just as in the file system analogy. Terracotta's data can be replicated, backed up, and otherwise managed just like a file server. Thus, the application developer can use objects in Java just like simple file I/O, and Terracotta will move the objects from the process to a central location where those objects can be managed safely and cost-effectively.

The last parallel between files and Terracotta that we mentioned was that the two enable clustering through simple, straightforward sharing of data and locking events. Applications that write to files by locking the parts of the file they want to work on can safely be shared across users in real time. Without NFS or similar networked file sharing technology, this simple approach will not work. The developer would have to push data among application processes in the application layer using network sockets by hand far above the file system. As a result, operators would lose the ability to manage these clustered applications by managing only their storage.

Files are a good example of a transparent clustering service. Developers write inside a file as they desire, while operators store that file as they desire. In short, transparent services, such as NFS, help developers write applications as they would prefer, while allowing operators the freedom to build scalable, highly available production environments for managed data.

At this point, you might be asking yourself, "What's the difference between the NFS protocol and the database?" With databases, developers use SQL, whereas with NFS, developers use the proprietary file layout. Also, with the database, the instance can run on any server we want without changing the application, and the database can be tuned without changing the application. The database certainly is a service. We are not asserting that NFS and files are better than SQL and a database. We are merely looking for good parallels to Terracotta. In its operating character, a database is a good parallel. In its programming model, however, it is not.

A developer knows when writing to files is necessary and does so intentionally. In the modern application, much of the use of the database is through an object-relational mapping tool like Hibernate, and thus, a developer who would rather not write to a database is forced to do so.

Terracotta works more like file I/O than a database, because Terracotta presents itself in the manner the developer desires. Terracotta is consumed as memory inside the JVM. The notion of migrating memory from the local server to a service somewhere on the network, as can be done with files, is quite a leap of faith. Nonetheless, Terracotta is a new example of a transparent service.

At its core, Terracotta's goal is to allow several computers to communicate with each other as if they were one big computer through memory and pure memory constructs, such as threads and locks (mutex objects and the like). You might be tempted to think of Terracotta as functioning just like NFS, because Terracotta takes a local resource and moves it to the network. The similarities to NFS start an end there, however. Let us now explore the power in having a transparent memory location in more detail and learn how it is unique and important.

Transparency of Memory Location

Unlike when working with files, developers using Terracotta do not see an API. Recall that our definition includes the notion of JVM-level clustering. This means that to the application, Terracotta is the JVM, and to the JVM, Terracotta is the application. Figure 1-1 illustrates this concept.

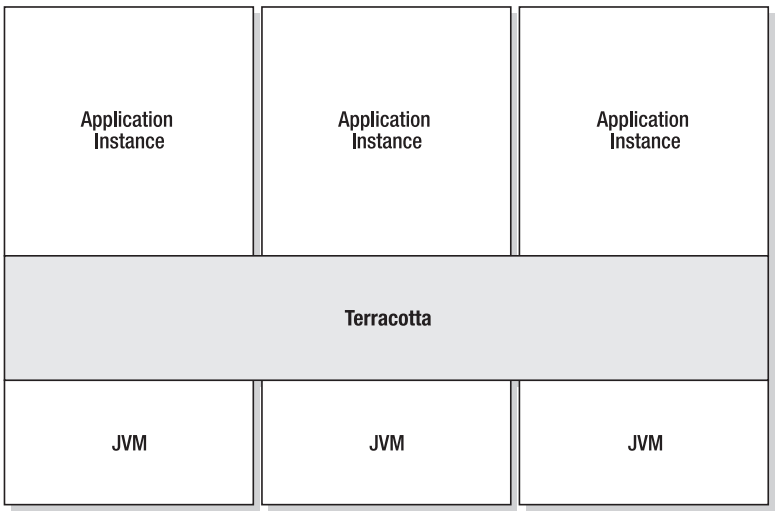

Figure 1-1. *Terracotta sits between the application and the JVM.*

Unlike file I/O, transparency of the memory location implies that Terracotta has to figure out what to share and move on its own. Think back to the file analogy and focus for a moment on the `open()` API call: `open(<path_to_file>, mode)`. A developer decides where to write the file relative to the application's running directory. A developer who wants an application to write to a file usually makes the file name and directory in which it resides a start-up option for the user. Suppose our application wants to write to a data file named `application.dat` in the directory path `/path/to/my/application/data`. The application merely needs to invoke `open("/path/to/my/application/data/application.dat")`. An operator can then move our application's data using NFS. To use NFS, the production operations team needs to mount the remote file system underneath the directory the application writes to when that application calls `open()`. In our example, this would look something like the following:

```
mount //remoteserver/remotepath /path/to/my/application/data
```

Once the remote mount is established, our application will write its data to the remote server.

With Terracotta, however, everything works at a memory level. The interface, in this case, is the object orientation of the language itself. The parallel to mounting remote repositories applies, because objects have structure just like files and directories. Developers build object models, and the JVM maps those objects onto the heap at run time. Object models are the key to interfacing with Terracotta, because they contain all the information the JVM needs to map an object model to memory, and thus, all the information Terracotta needs at the same time.

To explore this idea further, consider that object models are made up of graphs of related objects. A user has a first name, last name, and ID. These are its fields. A home address and work address can be stored inline in a user object as fields, but they are usually broken out into their own address class. Users can also have friends. This simple object model is illustrated in Figure 1-2.

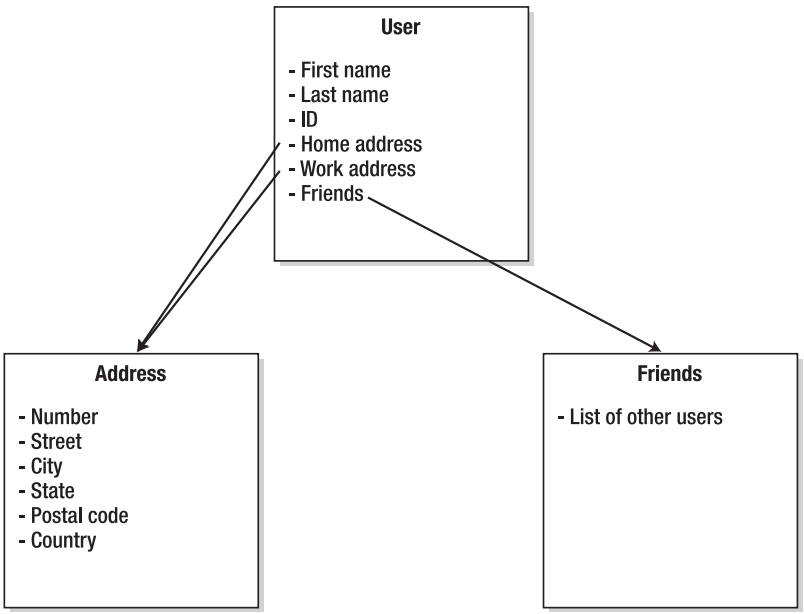

Figure 1-2. *A simple object domain model*

If a developer points to the `User` object, the expectation would be that the object references to `Home address`, `Work address`, and `Friends` would become shared automatically. In Terracotta, the developer need only specify the `User` object as being shared. Terracotta learns from the object definition itself what the fields of our shared object are (`First name`, `Last name`, etc.) as well as what the object relationships (`Addresses` and `Friends`) are. Consider this to be analogous to mounting a remote file system onto the local machine: the `User` object is mounted into memory from Terracotta, and after that, the memory operations can be cached locally but persisted remotely on the Terracotta service for sharing and later retrieval.

You will learn more about how this works in Chapter 3, but for now, know that Terracotta's ability to understand the intent of an application by analyzing the use of memory is what makes Terracotta transparent to our source code.

Putting Transparency and Clustering Together

Terracotta's transparent clustering service helps a developer share part of a Java process's memory across process boundaries and machines. This means that any object resident in memory in one Java process can be seen by another Java process, whether that process is on the same machine or somewhere else across the network. In the previous section, you learned that sharing part of a JVM means that we are mounting objects that appear local but are, in fact, stored in a central service. We should break this down in more detail.

First, let's build a simple class, `ClusteredClass`:

```
class ClusteredClass {
  int id;
  String name;

  ClusteredClass( int id, String name ) {
    this.id = id;
    this.name = name;
  }

  final Integer getid( ) {
    return new Integer( id );
  }

  final String getname( ) {
    return name;
  }

  void setid( int id ) {
    this.id = id;
  }

  void setname( String s ) {
    this.name = s;
  }
```

```
  public String toString( ) {
    return ( "Customer id=[" + id + "] name=[" + name + "]\n" );
  }
}
```

Now, imagine that there are two servers, and we want those two servers to act together as one logical server. We have to be assured by the JVM that instances of `ClusteredClass` created in one server are available to the other under the same object ID. Why? Simply put, if an object created on one JVM were not the same object on all JVMs, the application would not reap any value from transparent clustering. Think back to the NFS analogy. If two application instances wanted to read from and write to the same file, they must mount the same remote file system and open the same file by name. In Java, an object ID is like a file name. Terracotta uses the object ID as the clusterwide path to a specific object in memory.

Without consistent object identifiers across JVMs, the application would find that any object created on one server would be different and have to be re-created or reconnected to object graphs when moving from one JVM to another. Somehow, all this fuss does not sound transparent. Let's, therefore, assume that objects created with a particular ID in one JVM will have the same ID in any and all other JVMs.

It would also be nice if the second server could see an instance of `ClusteredClass` constructed on the first server. However, there are several challenges inherent to our sharing goal. Objects created in one memory space will have different addresses in RAM, and thus different identifiers, than in another space. Not only do memory addresses not align across processes, but objects created in one memory space will eventually get modified, and those modifications have to make it from one server's memory to the other's. How can we make all this happen? It is, after all, quite a leap to assume that Terracotta can find all of our `User` objects (from the previous section) and share them on its own.

One way to think of this challenge is in terms of object scope. If an object `o` is not created or referenced at an appropriate level of scope, such that it can be accessed across threads or even by the same thread at some time later, the object is not shareable across servers. Method-local scope refers to an object whose life cycle is that of the method in which it is created; the object is created and destroyed within the method's scope. Method-local objects tend to be immediate garbage to the modern collector and are not, therefore, what we are after. This is important: *sharing every object between the two servers is not necessary.* Sharing objects in collections, singletons, statics, and global objects is more what we are after. Server two should only be able to see object `o` that server one created if server one creates the object in a scope that is accessible to more than one thread. So if we invoke a constructor in a method-local context, we should not worry about clustering, but in the global context we should:

```
void printAClusteredClassInstance( ) { // method-level scope
  ClusteredClass c1 = new ClusteredClass( 1, "ari" );
  c1.toString( );
}

ClusteredClass clone( ClusteredClass c ) { // calling-level scope
  ClusteredClass c2 = new ClusteredClass( c.getid(), c.getname( ) );
  return c2; // c2 escapes the method so it might be interesting…
}
```

```
static Map myMap = new HashMap();

void storeAClusteredClassInstance( ClusteredClass c3 ) { // Stored in a singleton…
  myMap.put( new Integer( c3.getid( ) ), c3 );
}
```

In the preceding examples, `c1` is not worth clustering; `c2` might be worth clustering, depending on what happens in the thread that calls `clone()`; and `c3` should be clustered, because it joins `myMap`, which would likely be clustered. Thinking about it further, consider that if we are trying to get `c1`, `c2`, and `c3` shared between servers one and two. `c1` cannot, in fact, be shared because if server one creates it, there is no way for server two to reach `c1`. Similarly, there is no way for another thread in server one to reach `c1` (unless we change the declaration of `c1` to make it a static variable). `c2` might be clustered if the thread calling `clone()` actually places `c2` in a location in memory where multiple threads could reach it. `c3` is definitely and easily reachable by server two, however, when server two executes `myMap.get()` with the appropriate key.

Now, we have to examine the constructor, because Terracotta usually has to do something with constructor calls to get a handle on objects when those objects are first created (the constructor is sort of like the call to the mount command in the NFS analogy). Terracotta can decide when to invoke the constructor sort of like an automatic file system mounter. Let's now walk through an example.

■ **Note** `automount` first shipped with the Solaris Operating System. It detects the need for a remote file system and mounts that file system on demand.

In `storeAClusteredClassInstance()`, if we assume server one starts before server two, then we want only the first JVM to construct `myMap`, just like the code fragment would behave in one JVM. You are starting to see that Terracotta cannot only look for objects in memory but, in fact, has to address a few special instructions such as object construction, which translates into new memory allocation, as well as thread locking, which translates into mutex operations in the underlying operating system. In Chapters 3 and 4, we will explain how this works in more detail. For now, it is sufficient to know that there are only a handful of instructions that the JVM executes when manipulating memory, and Terracotta can observe all of them.

To round out our initial understanding of sharing objects across Java processes to achieve transparent clustering, threading and coordination must be considered. In the file analogy, an application process must lock the part of the file it wishes to write so that other processes do not corrupt it. For threads and locks, Terracotta can provide for cross-JVM locking just by using Java synchronization. You need to understand a bit about the Java memory model before you can fully understand this concept.

Threads are part of the clustering equation in that they manipulate objects and have to be synchronized when accessing those objects. Let's assume for a moment that Terracotta is deployed and working such that we have `c3` on servers one and two and that `c3` is exactly the same object in both JVMs. If one server wants to change `myMap`, it needs to lock `myMap`. Thus, locking and threading semantics must spread across processes, just like objects spread across Java processes.

Honoring the Java Memory Model

You know just about all you need to know to get started—except we still need to consider the memory model. We must synchronize our changes to `myMap` or `c3` across Java processes just as we would across Java threads; at least, we do if we want changes to either object to be sent reliably between servers one and two. In a sense, the process boundaries have melted away. But when do the servers communicate? And what do they send? The memory model provides the answer.

The Java Memory Model is quite well documented elsewhere, but the key concept we should focus on is "happens before," as laid out in JSR 133. To the JVM, "happens before" is defined as the logical time before a thread enters a synchronized block. The JVM assures that the objects this thread will gain access to once inside the synchronized block will be up to date.

■ **Note** The Terracotta authors relied on the graduate thesis of Bill Pugh at Carnegie Mellon University dated June, 2004 in designing the software, as did we when writing this book.

The model asserts that any changes that happened before a thread is granted entry into a synchronized block will be applied to main memory before the lock is granted. Terracotta uses the memory model to ensure proper cross-process behavior and semantics. For example, if object `o` has been changed by server one at some point in the recent past, server two will get those changes before entering into a synchronized block that is protecting access to `o`. Server one's changes in this case *happened before* server two began working with `o`, so server two sees server one's changes. Since locks are exclusive and pessimistic, meaning only one thread is allowed into a synchronized block at a time, the memory model proves sufficient for defining ordering in an environment where threads are spread across Java processes. We will go into more detail on this assertion in Chapter 4.

■ **Note** To the highly informed reader, Martin Fowler's statement "remember the first law of distributed computing: don't distribute your objects" might seem to be questioned by Terracotta's transparency. In fact, it is. Fowler's claim is too simple and ignores the fact that distributing objects is indeed ambiguous. Fowler was focused on remote method invocation (RMI). RMI serializes an object, sends it to a remote JVM, deserializes that object, runs the specified method, serializes the object, and returns control back to the first JVM. Terracotta does, in fact, distribute object data and coordinating events, but Terracotta never moves the processing context from one JVM to another as RMI would. Just as we can remotely store the files underneath an application without breaking that application, we can remotely store application objects without breaking the application. Therefore, Terracotta is not bound by Fowler's law as stated, because Terracotta never uses RMI, serialization, or other frameworks—each of which is not transparent. In fact, such frameworks throw exceptions that only the developer can address in source code, such as `RemoteMethodInvocationException`.

Being a Service Has Advantages

It has already been established that the technology offers a Java developer transparent clustering. Let's now turn our attention to the "service" component of our definition of Terracotta. Terracotta is made up of only a few components. It feels and smells like the services that operators are used to, meaning that Terracotta is not only some sort of agent or library running inside existing Java processes but also itself is a Java process.

To restate, Terracotta is made up of libraries you install next to any JVM that is to join a cluster, and Terracotta is a separate Java process unto itself, as shown in Figure 1-3.

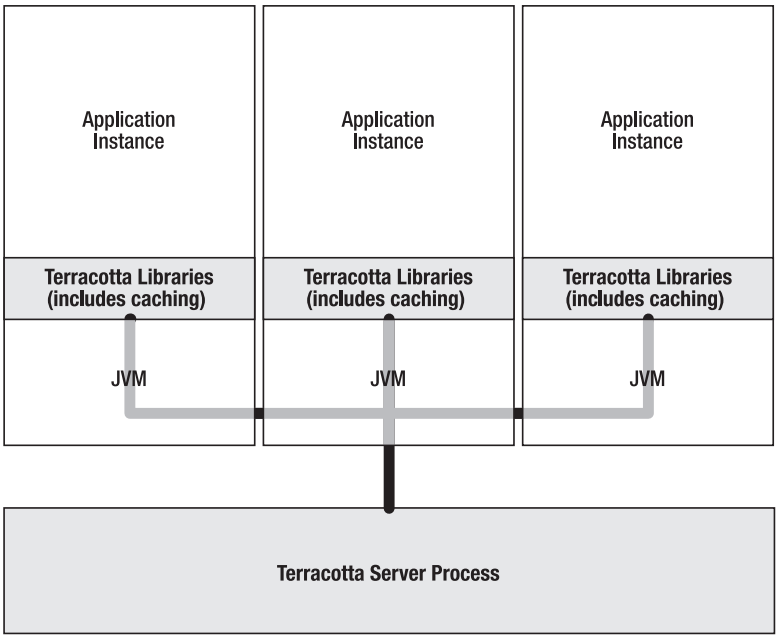

Figure 1-3. *Terracotta's two components include libraries and a server process that communicate over TCP/IP.*

The Java process running the Terracotta server runs *only* Terracotta. This Java process is the focal point of an operator's effort in managing a cluster of Java processes running on top of Terracotta. This means two things: scalability can be as easy as adding more JVMs, and high availability works similarly to that of the database or a file server. While applications use Terracotta to deliver availability and scalability, Terracotta itself delivers both scalability and availability when the operator managing the process can make the Terracotta service highly available. To simplify this concept, Terracotta makes an application highly available and Terracotta is, itself, made highly available much like a relational database.

Availability

In subsequent chapters, you will learn how Terracotta stores a copy of our shared object data outside our existing Java processes. This implies that if all our Java processes were to stop, we

could, in fact, start the processes again and gain access to those objects as they existed in memory before Java was terminated. When running on top of Terracotta's transparent clustering service, an application does not stop running until the operator flushes the objects from Terracotta's memory and disk. This is how Terracotta delivers high availability as a runtime service.

The Terracotta process is designed to be reliable and restartable. If it were killed or stopped, Terracotta could restart where it left off, and the application processes would not see any errors or exceptions while Terracotta restarts. The ability to restart is helpful for small applications and in development environments. For production systems, a more nonstop approach is preferable. Terracotta server processes can be configured to identify each other on the network and to share the duties of object management for an application cluster. In essence, Terracotta doesn't just cluster your application but is also capable of being clustered for availability using built-in features in the software.

Scalability

In subsequent chapters, you will also learn how to use Terracotta to tune and scale applications that are spread across Java process boundaries. Tuning applications can be achieved on multiple levels, from limiting the scope of the clustering behavior purely in Terracotta configuration files to sometimes making changes to source code. Source code changes can result in the biggest bang for the buck in terms of getting more scalability from an application running on Terracotta. We will study in detail how to configure Terracotta, as well as how to write code that is best suited for the environment.

Once an application is perfectly configured, or is at least as good as it is going to get, there is still more that can be done. If scaling is defined as adding more JVMs, and throughput is defined as how many operations per second a single node can handle, Terracotta can help in delivering more throughput per node as much as it can help in delivering better scaling to apply against the business need.

To help a JVM perform, Terracotta provides built-in caching designed to allow applications to perform at nonclustered speeds even when clustered. To improve performance, an application needs to optimize its use of Terracotta's internal caches. Optimizing the use of cache is a function of optimizing workload routing. One example of workload routing is sticky load balancing for web applications. In sticky load balancing, the same user session gets routed to the same JVM for every request. If some sort of stickiness can be applied, Terracotta will keep objects in memory as long as it can. Load balancing is not always as simple as stickiness and session replication, though. In general, developers and operators have to work together to consistently route a workload to the same JVM over time.

■ **Note** Several members of the Java community refer to Terracotta as a distributed cache. As you can see from the fact that Terracotta's caches are internal, and external load balancing is required to optimize cache efficiency, Terracotta is not a cache. It merely *uses* caching technology to ensure application performance and scalability.

By routing requests to the same JVM over time, we ensure that all the information a thread needs to process application requests is local to that JVM in memory. Without such assurance, our application will need Terracotta's server process to move the memory into context underneath us just in time. The application will have to wait for the network in order to complete an operation. This is commonly referred to as paging. Terracotta works at a byte level and not at a page level, but it has adopted the paging terminology from virtual memory inside operating systems because the notion is the same. In most operating systems, virtual memory allows an application to spill 4-kilobyte pages of memory to disk. The ability to page out unused memory frees the application to get more work done with less memory. When the application needs a page back from disk, that application begins to perform very slowly, however.

Here's another way to view the performance optimization Terracotta delivers through its use of caching inside the application's JVM: Terracotta will move as little data as possible given the changing workload it sees. This improves performance by allowing applications to move at the speed of memory instead of the speed of the network. Just like in the operating system and virtual memory, moving this caching feature into the transparent clustering service provides operators the configuration hooks to make runtime decisions about how large Terracotta's cache will be.

Terracotta can scale independently of the application above it. Much like a relational database, the Terracotta server's I/O and throughput are tuned using various IT strategies with which operators are familiar. These include disk striping and data partitioning. In a future release, Terracotta's designers also intend to provide the ability to cluster Terracotta servers for scale. Once the feature is implemented, Terracotta is able to stripe objects across Terracotta instances on its own.

Going back to our example code, Terracotta's core architecture can support storing `myMap` on one Terracotta server while storing `c3` on a second Terracotta server. When the application runs `myMap.get(id)`, Terracotta knows that the map is on one Terracotta instance, while `c3` is on another, and routes I/O based on Terracotta object identifier and corresponding routing information. Terracotta honors object identity as well as the Java Memory Model so there are no copies of objects, only real objects, which allows the system to store parts of objects on one Terracotta instance and other parts on another. This architecture delivers the ability to scale linearly, because any number of Terracotta instances will split the responsibility of storing and manipulating objects evenly: two instances split the data in half, three instances in thirds, and so on.

Avoiding Bottlenecks

The hardest part of tuning an application is in storing data on disk for availability yet simultaneously avoiding I/O bottlenecks. Figure 1-4 illustrates why tuning is hard. Essentially, the database helps us deliver high availability but with high latency or low scalability compared to purely in-memory applications. Simultaneously, caching and purely in-memory applications provide low and sometimes zero availability, since nothing is written to disk.

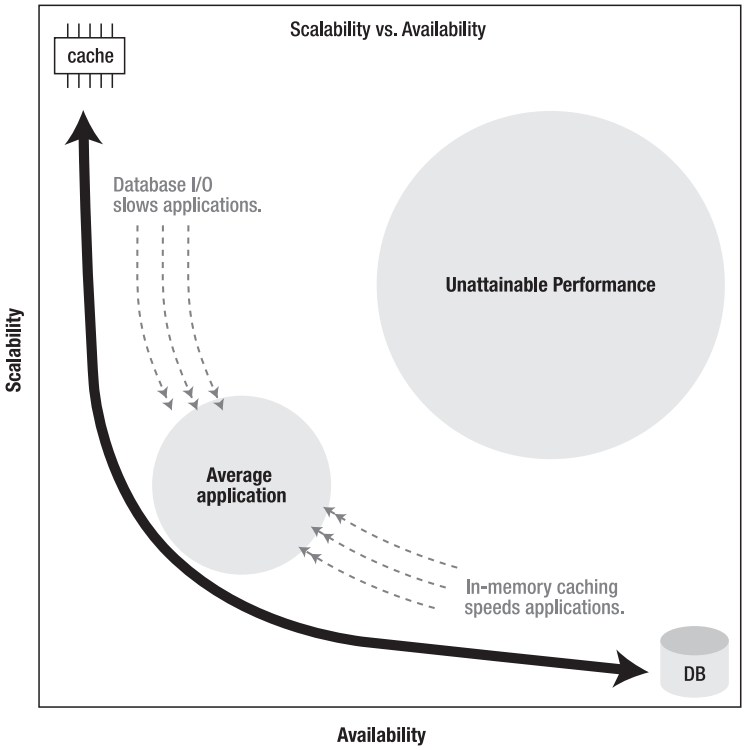

Figure 1-4. *Consider the scalability versus availability trade-off to help avoid bottlenecks and deliver reliability.*

The database can be mostly transparent through Hibernate and object-relational mapping, and it can support clustering if all application servers read and write to the same database tables. So why is the database not a transparent clustering service? The database should not be used for clustering, because it shifts the burden of scaling and clustering an application from application source code to production operations through complex and expensive database clustering technologies, such as Oracle Realtime Application Clusters (RAC). This is another way of saying what some of us might already know—the database is a bottleneck.

Unlike the database, Terracotta does not merely shift the object-sharing burden from the developer and source code to the operator. Terracotta's architecture and implementation contain several optimizations that provide an application with multiple lines of defense against disk I/O bottlenecks, the sum of these techniques make the solution over ten times faster than the database. Here are a few of these optimizations:

- Terracotta reads from in-memory caches when available.
- Terracotta reads only what it needs, and no more, from its server, and it can read in near-constant time from disk.
- Terracotta writes only what changes, and it writes the changes in batches.
- Terracotta writes in an append-only fashion. Objects are never sorted, indexed, or relocated/defragmented on the disk.

Let's analyze each of the four optimizations further, starting with reading from cache. First and foremost, the application has an in-memory cache of the data (see Figure 1-3) so that when reading, there is no disk I/O. This cache is on the application's side of the network with respect to the Terracotta server, so unlike a database's internal cache, Terracotta can respond to some requests at in-memory speed. Caching inside the application in a transparent fashion is only possible because Terracotta is transparent. This also serves to significantly lessen the workload on the Terracotta server process.

Second, Terracotta reads only what it needs. Terracotta stores an object's fields separate from each other on disk but near each other on the same disk block. Each field can be retrieved separately. Since the data is formatted on disk much like a `HashMap` would be in memory, Terracotta can take the identifier for a field to be retrieved as a key, hash that identifier into a file location, move to the appropriate block within the file, and load the field without loading any other disk blocks.

There is a further optimization here in that an object's fields are created at the same point in time that the constructor is called inside the application. Thus, the object's fields are all stored near each other on disk. Since Terracotta reads entire disk blocks at once, it usually reads, in a single disk operation, many objects and fields that were created together. Those objects and fields are often needed together, since they represent parts of the same object or closely related objects.

Third, the server writes what changes, and it writes in batches. As we pointed out, the Terracotta server writes to disk much like a hash map (`HashMap`) would write to memory. Each field can change without updating any other data. When several objects change, their changes are written together to the same block on disk so that they honor the block-level read optimization we discussed previously.

Fourth, the server writes in append-only fashion. This means that we needn't burden ourselves with changes in field data, and we needn't read blocks from disk in order to write blocks back to disk. Normally, changing data on disk requires reading the block where the data is located into memory, mutating the file buffers, and flushing the buffer back to disk. Terracotta avoids this extra read I/O. What's more, if the server running the Terracotta process has enough memory, the file data can all be cached in memory inside the operating system's file buffers. If this caching can occur, all read traffic is served from memory buffers (an additional line of defense), and the disk subsystem is only writing to the end of our data file. This implies that a Terracotta server can be tuned to utilize the entire disk subsystem's throughput, since it never has to seek to random locations on disk.

■ **Note** If a Terracotta server needs to support more write traffic than a single disk can sustain, the operator can stripe Terracotta across multiple disks so that write traffic spreads across those disks. For example, if a single hard disk spinning at 10,000 rpm can sustain 100 megabytes per second of read or write traffic, striping four disks together under Terracotta would deliver a total capacity of 400 megabytes per second of highly available object I/O to Java processes running on top of Terracotta. 400 megabytes per second translates to over 4 gigabits of network throughput.

These scaling and disk I/O recipes, while quite standard fare for database administrators, were brought to the attention of Terracotta's designers at the large e-commerce web site and are now baked into Terracotta

So, while the database can append new records onto the end of a table like Terracotta does, indexes must be adjusted and maintained as data changes. Databases must be prepared for random I/O into sorted indexes, and this costs significant latency while the disk seeks the point on the disk where an index must change. Terracotta, however, does not burden itself with where on the disk an object is stored relative to other objects. It merely needs to be able to retrieve individual objects by their IDs. Terracotta's singular use of the disk, as compared with the multiple uses a database has for the disk, means Terracotta can optimize itself, whereas a database requires administrator intervention to tell the system which of many optimizations to leverage for specific data.

Use Cases

Terracotta makes several servers work together like one logical server. It can be applied in many ways and to many use cases, but it works in conjunction with the JVM to eliminate scalability/availability trade offs that we otherwise deal with application after application.

The basic use cases of Terracotta fall under the following umbrella categories:

- Distributed cache
- Database offload
- Session replication
- Workload partitioning

Distributed Caching

When used to distribute application cache data, Terracotta shares Java collections across Java processes. Under such architectures, typical cases require maps, lists, arrays, and vectors—in both `java.util` and `java.util.concurrent`—to be shared across processes. There are several Terracotta capabilities that will make this use case easier. The first is that object identity will work. This means that `hashCode()` and `==` will work as expected, for example, in maps and lists with respect to storing the same key more than once. This also means that the following code example will return `TRUE`:

```
Object foo1, foo2;
foo1 = cache.get( key );
foo2 = cache.get( key );
return( foo1 == foo2 );
```

Another benefit of transparent clustering services for distributed cache applications is that the cache does not impinge on the domain model. The classic example here is that of multiple indexes to a dataset. For example, a web site in the travel industry needs to be able to display hotels by location, price, room amenities, and availability. The natural place to store the hotel data is in a database where all these details amount to SQL queries against a data table and indexes are specified at run time. However, most travel web sites are not issuing

queries against the database every time a user comes to the site and attempts to locate a hotel; rather, they use a cache. And, it would be nice to be able to do in the cache what we can do in the database, namely store the hotel once and only once but have alternate indexes with price ranges, location names, and other data as different keys.

I can store the Westin St. Francis Hotel in downtown San Francisco under the keys "San Francisco," "US POSTAL CODE 94102," and "Hotels between $300 and $1,000 per night". But the hotel is the same hotel no matter how it gets returned. This is all fine and good until we want to update the hotel's price to $259 on special; now, the hotel falls out of the price range under which it was originally stored in cache, because it is now less than $300. If we model this in the database, there's no problem. If we model it in memory, spread across machines—problem. Chapter 5 will detail the distributed cache use case and how to work with changing data over time.

Database Offload

The database is the bane of Java developers' existence and, simultaneously, our best friend. The key reasons for using a database are that the data gets written to disk and can be queried and updated in a well-known format and that the database can be tuned without significant impact to the business logic. The reasons not to use the database are that the object-relational translation complicates code, transactional architectures are hard to scale, and the database is a bottleneck.

Applications can use Terracotta to eliminate the database bottleneck. In such scenarios, leave objects in memory, and cluster them via Terracotta. Since Terracotta is infrastructure separate from any application process, objects in memory do not necessarily get lost (recall that Terracotta stores objects on disk, so objects will not get lost). Therefore, when using Terracotta, the database's role is well contained; it is used primarily for query and reporting of business data. It is used less as a central store for objects on disk and as a transactional control point where synchronization suffices.

Session Replication

Session replication has long been a feature of the application server or container. Apache Tomcat, BEA WebLogic, JBoss Application Server, IBM Websphere (CE), Jetty from Webtide, and Sun GlassFish each has a session replication solution bundled within the container. But transparent infrastructure for sharing objects across processes can provide unique value when compared to these frameworks.

Terracotta's transparency implies something new and different from all prior session replication approaches. Specifically, session replication usually pushes a few programming rules back up through the container to the application developer. Web application developers know that all session attributes must implement `java.io.Serializable`. Developers also know that those attributes must be small. And domain objects should not be referenced by the session attributes, or you risk serializing too much or serializing dangerous data. Last, any attribute that is accessed via `getAttribute()` must later be put back into session by calling `setAttribute()`.

These rules and more regarding the use of session replication are, in fact, direct results of containers' having used session serialization and replication when sharing objects in session. If we follow the session, what we would find is that the session is retrieved from an internal `HashMap` when the container receives a web request. That session is marked as clean when the

container hands the session object to the servlet code. The session will get serialized and sent across the wire to other instances of the application only if we call `setAttribute()` when changing objects inside the session.

Transparent object sharing across Java processes is different. Transparent object sharing implies that if we, the application developers, can share the session `HashMap` and all the sessions and attributes inside that map, then we needn't signal to the container what we change and when. We no longer need to call `setAttribute()` when we change attributes. Nor do we have to make sure objects implement the serializable interface, nor worry about their size. In some cases, references to domain objects and singletons will even prove acceptable.

Workload Partitioning

Workload partitioning is currently referred to under more popular names like MapReduce, data grid, and compute grid. The idea here is to divide and conquer. With Terracotta, divide-and-conquer architecture is achieved either by using Java data structures for messaging along with thread coordination or by using JMS or socket communications instead of threading for coordination and using shared data structures for a different purpose.

Without going into too much detail now, Figure 1-5 illustrates the power of partitioning. Essentially, an application can spread computational workload such as querying, updating of large amounts of data, or similar use cases without the need for message queuing, a database, or other infrastructure services. A Java application and some multithreaded logic will suffice.

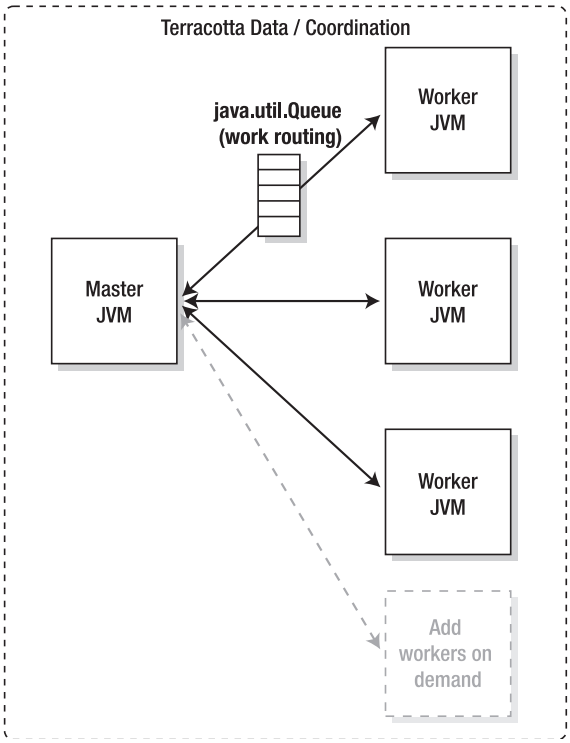

Figure 1-5. *Distributing workload to many computers using only memory, threads, and Terracotta (see Chapters 10 and 11 for details)*

Summary

Terracotta is a transparent clustering service with an object-oriented development model. "Transparent" refers to the pure Java language and JVM memory interface with no explicit programming model requirements, such as a requirement to use Spring or Hibernate. "Clustering" refers to the ability to allow multiple machines and Java processes to work together on the same data and to signal each other using pure objects and threading logic. And "service" refers to the infrastructure nature of this technology. A look at distributed caching APIs helps tighten this definition.

Distributed caches are not transparent clustering services. These libraries are designed to push bytes among networked servers when the cache is updated through `get()` and `put()` calls, and it's important to note the following details about them:

- *Transparency*: Distributed caches cannot be transparent. They require the developer to call `get()`, which copies data from the cache into objects when the API is called. Distributed caches also require the developer to call `put()` after objects are changed; this copies data back to the cache. `get()` and `put()` imply that the application code and the object model cannot support certain things. Methods in our source code cannot hold on to references to objects across calls to `get()`, because we will not get the up-to-date data from the cache from the last time a thread called `put()`. And object models cannot include rich domains where objects, like our `User` and his `Address` and `Friends` from earlier in this chapter, could actually be Java references. If our domain model is made up of regular references, the entire object graph will get serialized on `put()`, which means we will end up with different objects on `get()` than we initially created.

- *Clustering*: Distributed caches deliver Java developers a form of clustering. They do not provide clustering as a service, as we defined "service" earlier, because they are not transparent. An application cannot be written for a single machine and then operate across a group of machines in production. Furthermore, since most clustered caching libraries must be present at runtime for even threads on one machine to share objects, they cannot be removed from the application without rewriting that application to some degree.

- *Simplicity*: The relative complexity of distributed caches is quite high, not just because they force changes to the application source but also because they require changes to the production operating procedure. This is due to the fact that, in order to scale, distributed caches ask that we keep all objects in memory spread across many JVMs. For example, if we want to manage a 100-gigabyte dataset, and our JVM is 32-bit, we will need a minimum of 50 JVMs at 2 gigabytes of heap per JVM to store the entire data set. If we want that data to be highly available, we need to store each object in two JVMs, meaning we need a minimum of 100 JVMs. 100 JVMs are required just to manage 100 gigabytes of data, and this example ignores the application's actual purpose and business workload, which will surely drive even more servers.

- *Scalability*: Distributed caches have historically scaled better than applications clustered using only a database by putting objects at risk (recall Figure 1-4, where we trade off scalability and availability) and keeping our object data only in memory. Having JVMs hold our objects only in memory means that we cannot eliminate the database, because a loss of power would otherwise imply a loss of data. Thus we end up in a hybrid world with clustering plus databases and their associated scalability issues, which is back where we started

By our definition, distributed caches are not transparent clustering services. Consider now that distributed caches are rich networking libraries where developers do not see network stacks and APIs, such as sockets. Because these caches hide the network transport, compared to TCP or UDP, distributed caches are a higher form of networking. The higher form is valuable, but it is neither transparent nor a clustering service.

The definition of Terracotta as a transparent clustering service presented in this chapter is something new and worth exploring in further detail: in Chapter 2, you will learn about the history of the concept, and in Chapter 3, we will dive into our first example—`HelloClusteredWorld`.