Chapter 9
Case Study Design

9.1 Overview

In this chapter, a design for the ICDE case study described in Chap. 2 is given. First, a little more technical background to the project is given, so that the design details are more easily digested. Then the design description is presented, and is structured using the architecture documentation template introduced in the previous chapter. The only section that won't be included in the document is the first, the "Project Context", as this is basically described in Chap. 2. So, without further ado, let's dive into the design documentation.

9.2 ICDE Technical Issues

Chapter two gave a broad, requirements level description of the ICDE v1.0 application and the goals for building the next version. Of course, this description is necessary in order to understand architectural requirements, but in reality, it's only the starting point for the technical discussions that result in an actual design. The following sections describe some of the technical issues, whose solutions are reflected in the resulting design description later in the chapter.

9.2.1 Large Data

The ICDE database stores information about the actions of each user when using their workstation and applications. This means events such as opening and closing applications, typing in data, moving the mouse, accessing the Internet, and so on all cause data to be written to the database. Although the database is periodically purged (e.g., every day/week) to archive old data and control size, some database tables can quickly grow to a size of several million rows.

I. Gorton, *Essential Software Architecture*,
DOI 10.1007/978-3-642-19176-3_9, © Springer-Verlag Berlin Heidelberg 2011

This is not a problem for the database to handle, but it does create an interesting design issue for the ICDE API. With the two-tier ICDE v1.0 application, the data analysis tool can issue naïve database queries (the classic *SELECT * from VERYBIGTABLE* case) that can return very large data sets. These are inevitably slow to execute and can bring down the analysis tool if the data set returned is very large.

While inconvenient, according to the "you asked for it, you got it!" principle, this isn't a serious issue for users in a single user system as in ICDE v1.0. They only do harm to themselves, and presumably after bringing down the application a few times, will learn better.

However, in order to lower deployment costs and management complexity, transitioning the ICDE system to be shared amongst multiple users is a potentially attractive option because:

- It reduces database license costs, as only one is needed per deployment, not per user.
- It reduces the specification of the PC that users need to run the ICDE application, as it doesn't need to run the database, just the ICDE client software. Simply, this saves money for a deployment.
- It reduces support costs and simplifies database management, as there's only one shared ICDE server application to manage and monitor.

If the database is to be shared by multiple users, it would still be possible to use a two-tier or three-tier application architecture. The two-tier option would likely provide better performance for small deployments, and be easier to build as less components would be needed (basically, no middle tier). The three-tier option would likely scale better as deployments approach a 100–150 users, as the database connections can be pooled and additional processing resources deployed in the middle tier.

Regardless, when a shared infrastructure is used, the behavior of each client impacts others. In this case, issues to consider are:

- Database performance
- For the three-tier option, resource usage in the middle tier

Memory usage in the middle tier is an important issue to consider, especially as ICDE clients (both users and third party tools) might request result sets with many thousands of rows. While the middle tier server could be configured with a large memory heap, if several clients request sizeable result sets simultaneously, this could easily consume all the servers memory resources, causing thrashing and poor performance. In some cases this will cause requests to fail due to lack of memory and timeouts, and will likely bring down the server in extreme cases.

For third party tools written to the ICDE API, this is not at all desirable. If potentially huge result sets can be returned from an API request, it means it is possible to create applications using the API that can fail unpredictably. The failure circumstances would depend on the size of the result set and the concurrent load on the server exerted by other clients. One API call might bring down the server, and

cause all applications connected to the server to fail too. This is not likely to make the development or support teams happy as the architecture would not be providing a reliable application platform.

9.2.2 Notification

There are two scenarios when event notification is needed.

1. A third party tool may want to be informed when the user carries out a specific action, for example, accesses a new site on the Internet.
2. The third party tools can share useful data that they store in the ICDE database with other tools. Therefore they need a mechanism to notify any interested parties about the data they have just written into the ICDE system.

Both of these cases, but especially the first, require the notification of the event to be dispatched rapidly, basically as the event occurs. With a two-tier architecture, instant notification is not so natural and easy to achieve. Database mechanisms such as triggers can be used, but these have disadvantages potentially in terms of scalability, and also flexibility. A database trigger is a block of statements that are executed when there is an alteration (INSERT, UPDATE, DELETE) to a table in the database. Trigger mechanisms tend to exploit database vendor specific features, which would inhibit portability.

Flexibility is the key issue here. The ICDE development team cannot know what events or data the third party tools wish to share a priori (simply, the tools don't exist yet). Consequently, some mechanism that allows the developers themselves to create and advertise events types "on demand" is needed. Ideally, this should be supported by the ICDE platform without requiring intervention from an ICDE programmer or administrator.

9.2.3 Data Abstraction

The ICDE database structure evolved considerably from v1.0 to v2.0. The reasons were to incorporate new data items, and to optimize the internal organization for performance reasons. Hence it is important that the internal database organization is not exposed to API developers. If it were, every time the schema changed, their code would break. This would be a happy situation for precisely no one.

9.2.4 Platform and Distribution Issues

Third party tool suppliers want to be able to write applications on non-Windows platforms such as Linux. Some tools will want to run some processes on the same

workstation as the user (on Windows), others will want to run their tools remotely and communicate with the user through ubiquitous mechanisms like email and instant messaging. Again, the key here is that the ICDE solution should make both options as painless as possible.

9.2.5 API Issues

The ICDE API allows programmatic access to the ICDE data store. The data store captures detailed, time-stamped information about classes of events of user actions, including:

- Keyboard events
- Internet browser access events
- Application (e.g., word processor, email, browser) open and close events
- Cut and paste events
- File open and close events

Hence the API must provide a set of interfaces for querying the event data stored in the database. For example, if a third party tool wants to know the applications a user has opened since they last logged on (their latest ICDE "session"), in pseudo code the API call sequence might look something like:

```
Session sID = getSessionID(userID, CURRENT_SESSION);
ApplicationData[] apps = getApplicationEvent(sID,
   APP_OPEN_EVENT, NULL); //NULL = all applications
```

The `apps` array can now be walked through and, for example, the web pages opened by the user in their browser during the session can be accessed[1] and analyzed using more API calls.

The ICDE API should also allow applications to store data in the data store for sharing with other tools or perhaps the user. An API for this purpose, in pseudo-code, looks like:

```
ok = write( myData, myClassifier, PUBLISH, myTopic);
```

This stores the data in a predesignated database table, along with a classifier that can be used to search for and retrieve the data. The API also causes information about this event to be published on topic `myTopic`.

In general, to encourage third party developers, the ICDE API has to be useful in terms of providing the developers with the facilities they need to write tools. It should therefore:

[1]The ICDE data store keeps copies of all accessed web pages so that even dynamically changing web pages (e.g., http://www.bbc.co.uk) can be viewed as they appeared at the time of access.

- Be easy to learn and flexibly compose sequences of API queries to retrieve useful data.
- Be easy to debug.
- Support location transparency. Third party tools should not have to be written to a particular distributed configuration that relies on certain components being at known, fixed locations.
- Be resilient as possible to ICDE platform changes. This means that applications do not break when changes to the ICDE API or data store occur.

9.2.6 Discussion

Taken together, the above issues weave a reasonably complex web of requirements and issues. The event notification requirements point strongly to a flexible publish–subscribe architecture to tie together collaborating tools. The need to support multiple platforms and transparent distributed configurations points to a Java solution with the various components communicating over protocols like RMI and JMS. The large data and data store abstraction requirements suggest some layer is needed to translate API calls into the necessary SQL requests, and then manage the safe and reliable return of the (potentially large) result set to the client.

The solution the ICDE team selected is based on a 3 three-tier architecture along with a publish–subscribe infrastructure for event notification. The details of this solution, along with detailed justifications follow in the next section, which documents the architecture using the template from Chap. 6.

9.3 ICDE Architecture Requirements

This section describes the set of requirements driving the design of the ICDE application architecture.

9.3.1 Overview of Key Objectives

The first objective of the ICDE v2.0 architecture is to provide an infrastructure to support a programming interface for third party client tools to access the ICDE data store. This must offer:

- Flexibility in terms of platform and application deployment/configuration needs for third party tools.
- A framework to allow the tools to "plug" into the ICDE environment and obtain immediate feedback on ICDE user activities, and provide information to analysts and potentially other tools in the environment.
- Provide convenient and simple read/write access to the ICDE data store.

The second objective is to evolve the ICDE architecture so that it can scale to support deployments of 100–150 users. This should be achieved in a way that offers a low cost per workstation deployment.

The approach taken must be consistent with the stakeholder's needs, and the constraints and nonfunctional requirements detailed in the following sections.

9.3.2 Architecture Use Cases

Two basic use cases regarding the API usage have been identified from discussions with a small number of potential third party tool vendors. These are briefly outlined below:

- *ICDE data access*: Queries from the third party tools focus on the activities of a single ICDE user. A query sequence starts by getting information about the user's current work assignment, which is basically the project (i.e., "analyze Pfizer Inc financials") they are working on. Query navigation then drills down to retrieve detailed data about the user's activity. The events retrieved are searched in the time sequence they occur, and the application logic looks for specific data items (e.g., window titles, keyboard values, document names, URLs) in the retrieved records. These values are used to either initialize activity in the third party analysis tool, or create an informational output that appears on the user's screen.
- *Data Storage*: Third party tools need to be able to store information in the ICDE data store, so that they can share data about their activities. A notification mechanism is needed for tools to communicate about the availability of new data. The data from each tool is diverse in structure and content. It must therefore contain associated discoverable metadata if it is to be useful to other tools in the environment.

9.3.3 Stakeholder Architecture Requirements

The requirements from the perspectives of the three major project stakeholders are covered in the following sections.

9.3.3.1 Third Party Tool Producers

- *Ease of data access*: The ICDE data store comprises a moderately complex software component. The relational database has approximately 50 tables, with

some complex interrelationships. In the ICDE v1.0 environment, this complexity makes the SQL queries to retrieve data nontrivial to write and test. Also, as the functional requirements evolve with each release, changes to the database schema are inevitable, and these might break existing queries. For these reasons, a mechanism to make it easy for third party tools to retrieve useful data is needed, as well as an approach to insulate the tools from database changes. Third party tools should not have to understand the database schema and write complex queries.

- *Heterogeneous platform support*: Several of the third party tools are developing technologies on platforms other than Windows. The ICDE v1.0 software is tightly coupled to Windows. Also, the relational database used is available only on the Windows platform. Hence, the ICDE v2.0 must adopt strategies to make it possible for software not executing on Windows to access ICDE data and plug into the ICDE environment.
- *Instantaneous event notification*: The third party tools being developed aim to provide timely feedback to the analysts (ICDE users) on their activities. A direct implication of this is that these tools need access to the events recorded by the ICDE system as they occur. Hence, some mechanism is needed to distribute ICDE user-generated events as they are captured in the *Data Store*.

9.3.3.2 ICDE Programmers

From the perspective of the ICDE API programmer, the API should:

- Be easy and intuitive to learn.
- Be easy to comprehend and modify code that uses the API.
- Provide a convenient, concise programming model for implementing common use cases that traverse and access the ICDE data.
- Provide an API for writing tool specific data and metadata to the ICDE data store. This will enable multiple tools to exchange information through the ICDE platform.
- Provide the capability to traverse ICDE data in unusual or unanticipated navigation paths. The design team cannot predict exactly how the data in the data store will be used, so the API must be flexible and not inhibit “creative” uses by tool developers.
- Provide “good” performance, ideally returning result sets in a small (1–5) number of seconds on a typical hardware deployment. This will enable tool developers to create products with predictable response times.
- Be flexible in terms of deployment options and component distribution. This will make it cost-effective to establish ICDE installations for small workgroups, or large departments.
- Be accessible through a Java API.

9.3.3.3 ICDE Development Team

From the ICDE development team's perspective, the architecture must:

- Completely abstract the database structure and server implementation mechanism, insulating third party tools from the details of, and changes to, the ICDE data store structure.
- Support ease of server modification with minimal impact on the existing ICDE client code that uses the API.
- Support concurrent access from multiple threads or ICDE applications running in different processes and/or on different machines.
- Be easy to document and clearly convey usage to API programmers.
- Provide scalable performance. As the concurrent request load increases on an ICDE deployment, it should be possible to scale the system with no changes to the API implementation. Scalability would be achieved by adding new hardware resources to either scale up or scale out the deployment.
- Significantly reduce or ideally remove the capability for third party tools to cause server failures, consequently reducing support effort. This means the API should ensure that bad parameter values in API calls are trapped, and that no API call can acquire all the resources (memory, CPU) of the ICDE server, thus locking out other tools.
- Not be unduly expensive to test. The test team should be able to create a comprehensive test suite that can automate the testing of the ICDE API.

9.3.4 Constraints

- The ICDE v1.0 database schema must be used.
- The ICDE v2.0 environment must run on Windows platforms.

9.3.5 Nonfunctional Requirements

- *Performance*: The ICDE v2.0 environment should provide sub five second response times to API queries that retrieve up to 1,000 rows of data, as measured on a "typical" hardware deployment platform.
- *Reliability*: The ICDE v2.0 architecture should be resilient to failures induced by third party tools. This means that client calls to the API cannot cause the ICDE server to fail due to passing bad input values or resource locking or exhaustion. This will result in less fault reports and easier and cheaper application support. Where architectural trade-offs must be made, mechanisms that provide reliability are favored over those that provide better performance.

- *Simplicity*: As concrete API requirements are vague (because few third party tools exist), simplicity in design, based on a flexible[2] foundation architecture, is favored over complexity. This is because simple designs are cheaper to build, more reliable, and easier to evolve to meet concrete requirements as they emerge. It also ensures that, as the ICDE development team is unlikely to possess perfect foresight, highly flexible[3] and complex, but perhaps unnecessary functionality is not built until concrete use cases justify the efforts. A large range of features supported comes at the cost of complexity, and complexity inhibits design agility and evolvability.

9.3.6 Risks

The major risk associated with the design is as follows:

Risk	Mitigation strategy
Concrete requirements are not readily available, as only a few third party tool vendors are sufficiently knowledgeable about ICDE to provide useful inputs	Keep initial API design simple and easily extensible. When further concrete use cases are identified, extend the API where needed with features to accommodate new requirements

9.4 ICDE Solution

The following sections outline the design of the ICDE architecture.

9.4.1 Architecture Patterns

The following architecture patterns are used in the design:

- *Three-tier*: Third party tools are clients, communicating with the API implementation in the middle tier, which queries the ICDE v2.0 data store.
- *Publish–subscribe*: The middle tier contains a publish–subscribe capability.
- *Layered*: Both the client and middle tier employ layers internally to structure the design.

[2]Flexible in terms of easy to evolve, extend and enhance, and not including mechanisms that preclude easily adopting a different architectural strategy.

[3]Flexible in terms of the range of sophisticated features offered in the API for retrieving GB data.

9.4.2 Architecture Overview

The ICDE v2.0 architecture overview is depicted in Fig. 9.1. ICDE clients use the *ICDE API Client* component to make calls to the *ICDE API Services* component. This is hosted by a JEE application server, and translates API calls into JDBC calls on the data store. The existing *Data Collection* client in ICDE v1.0 is refactored in this design to remove all functionality with data store dependencies. All data store access operations are relocated into a set of JEE hosted components which offer data collection services to clients.

Event notification is achieved using a publish–subscribe infrastructure based on the Java Messaging Service (JMS).

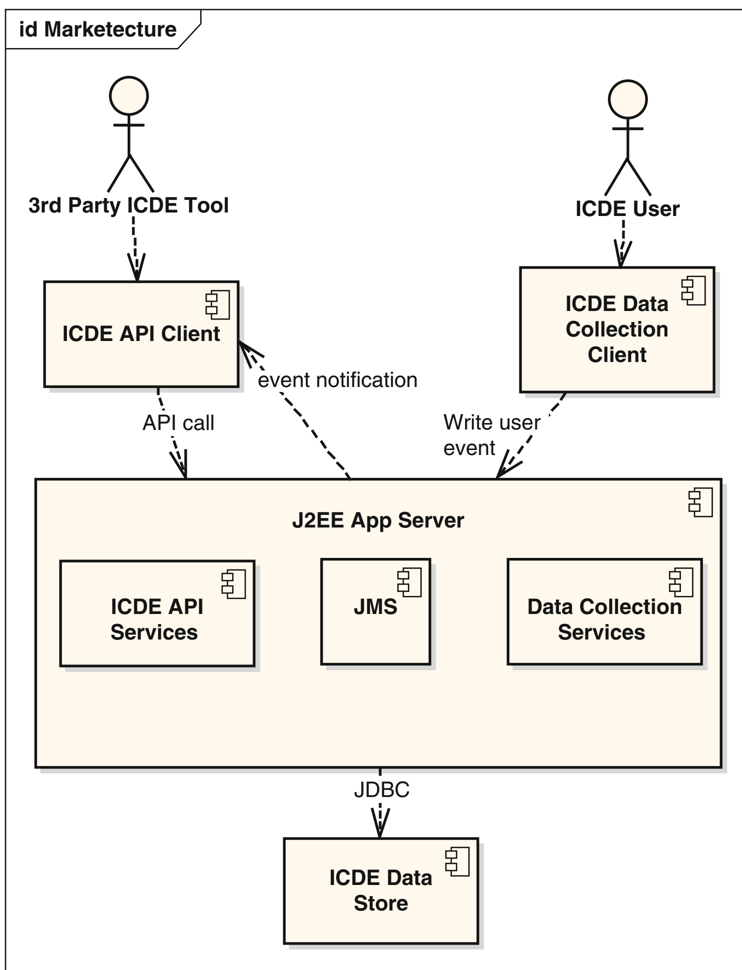

Fig. 9.1 ICDE API architecture

Using JEE as an application infrastructure, ICDE can be deployed so that one data store can support:

- Multiple users interacting with the data collection components.
- Multiple third party tools interacting with the API components.

9.4.3 Structural Views

A component diagram for the API design is shown in Fig. 9.2.

This shows the interfaces and dependencies of each component, namely:

- *ICDE Third Party Tool*: This uses the *ICDE API Client* component interface. The API interface supports the services needed for the third party tool to query the data store, write new data to the data store, and to subscribe to events that are

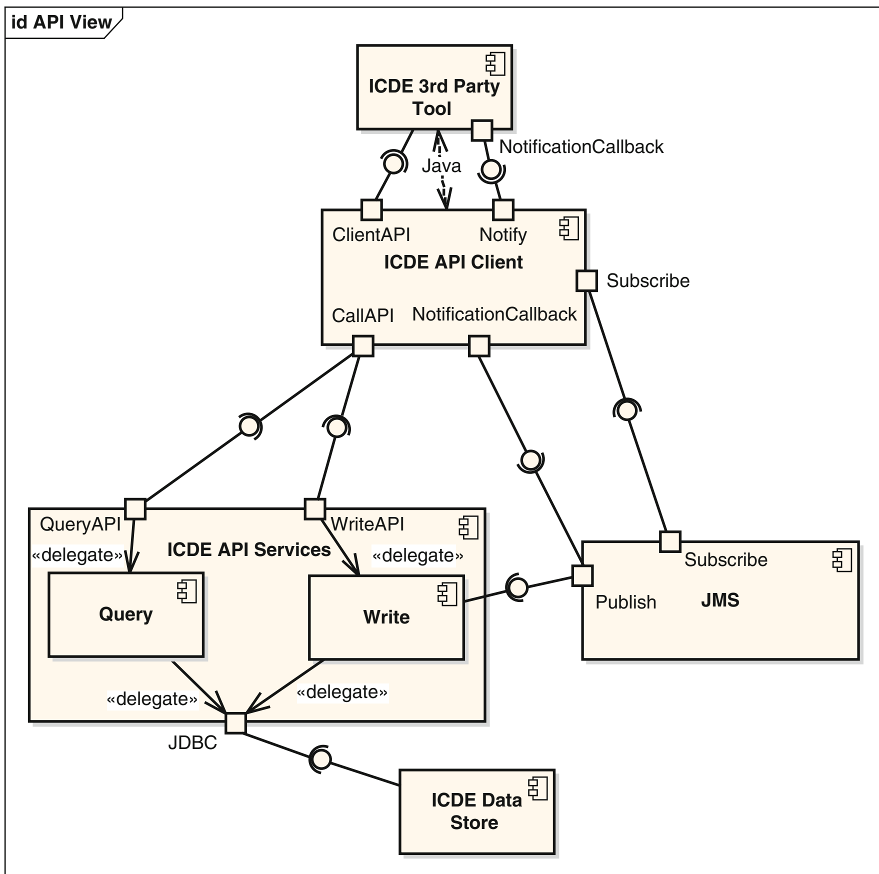

Fig. 9.2 Component diagram for ICDE API architecture

published by the JMS. It must provide a callback interface that the *ICDE API Client* uses to deliver published events.

- *ICDE API Client*: This implements the client portion of the API. It takes requests from third party tools, and translates these to EJB calls to the API server components that either read or write data from/to the data store. It also packages the results from the EJB and returns these to the third party tool. This component encapsulates all knowledge of the use of JEE, insulating the third party tools from the additional complexity (e.g., locating, exceptions, large data sets) of interacting with an application server. Also, when a third party tool requests an event subscription, the *ICDE API Client* issues the subscription request to the JMS. It therefore becomes the JMS client that receives published events, and it passes these on using a callback supported by the third party tools.
- *ICDE API Services*: The API services component comprises stateless session EJBs for accessing the *ICDE Data Store* using JDBC. The *Write* component also takes a topic parameter value from the client request and publishes data about the event on the named topic using the JMS.
- *ICDE Data Store*: This is the ICDE v2.0 database.
- *JMS*: This is a standard JEE Java Messaging Service, and supports a range of topics used for event notification using the JMS publish–subscribe interfaces.

A component diagram for the data collection functionality is depicted in Fig. 9.3. The responsibilities of the components are:

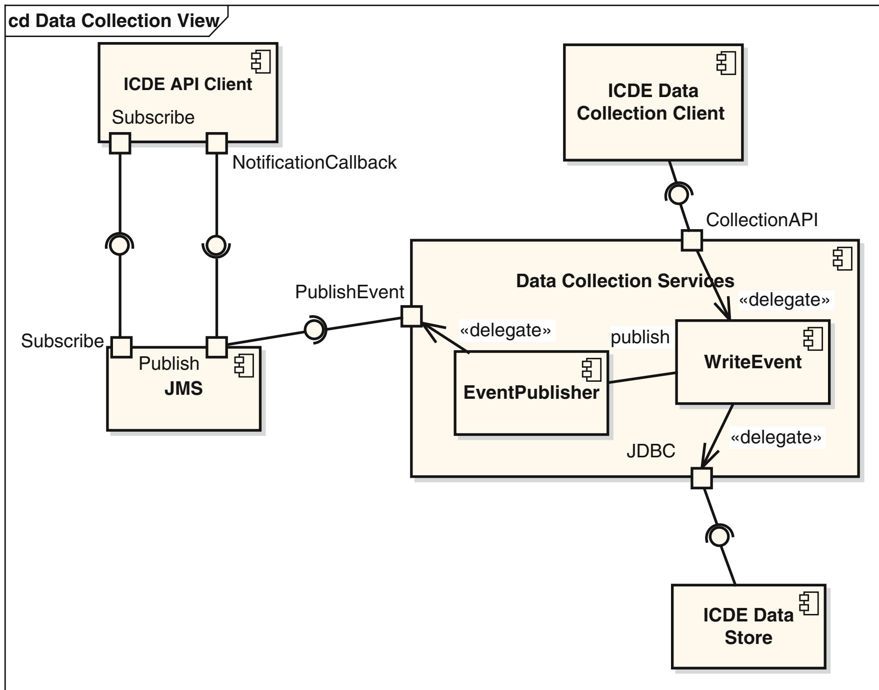

Fig. 9.3 Data collection components

- *ICDE Data Collection Client*: This is part of the ICDE client application environment. It receives event data from the client application, and calls the necessary method in the *CollectionAPI* to store that event. It encapsulates all knowledge of interacting with the JEE application server in the ICDE client application.
- *Data Collection Services*: This comprises stateless session EJBs that write the event data passed to them as parameters to the *ICDE Data Store*. Some event types also cause an event notification to be passed to the *EventPublisher*.
- *EventPublisher*: This publishes event data on the JMS using a set of preconfigured topics for events that should be published (not all user generated events are published, e.g., moving the mouse). These events are delivered to any *ICDE API Client* components that have subscribed to the event type.

A deployment diagram for the ICDE architecture is shown in Fig. 9.4. It shows how the various components are allocated to nodes. Only a single ICDE user and a

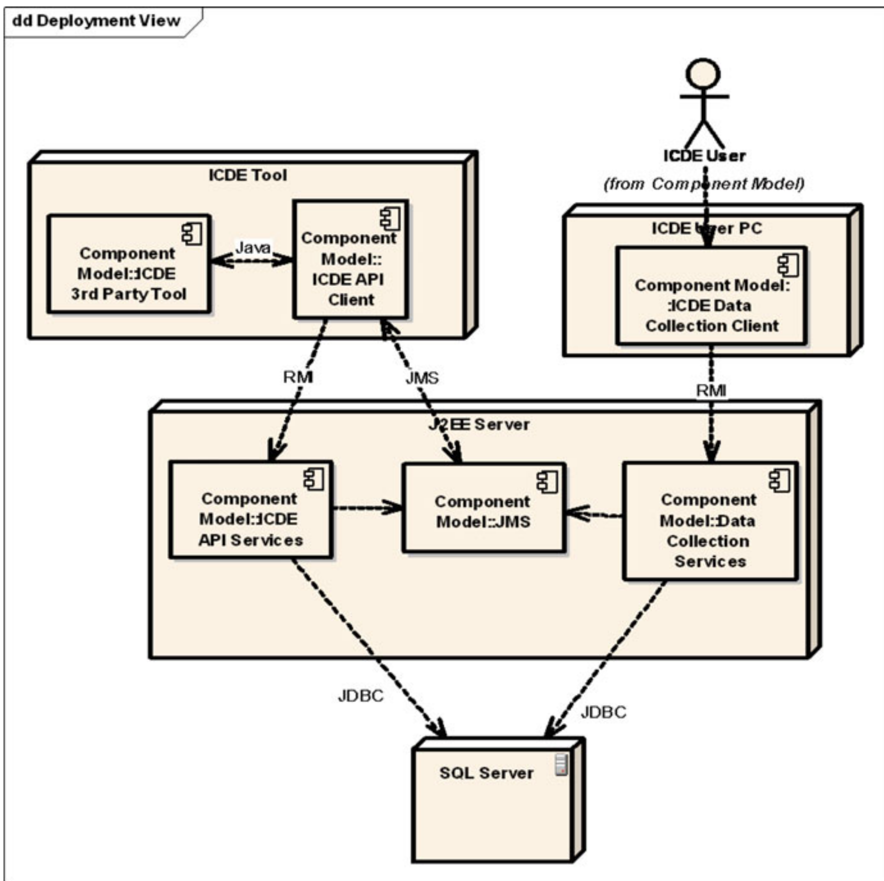

Fig. 9.4 ICDE deployment diagram

single third party tool are shown, but the JEE server can support multiple clients of either type. Issues to note are:

- Although the third party tools are shown executing on a different node to the ICDE user workstation, this is not necessarily the case. Tools, or specific components of tools, may be deployed on the user workstation. This is a tool-dependent configuration decision.
- There is one *ICDE API Client* component for every third party tool instance. This component is built as a JAR file that is included in the tool build.

9.4.4 Behavioral Views

A sequence diagram for a query event API call is shown in Fig. 9.5. The API provides an explicit “Initialize” call which tools must invoke. This causes the *ICDE API Client* to establish references to the EJB stateless session beans using the JEE directory service (JNDI).

Once the API layer is initialized, the third party tool calls one of the available query APIs to retrieve event data (perhaps a list of keys pressed while using the word processor application on a particular file). This request is passed on to an EJB instance that implements the query, and it issues the JDBC call to get the events that satisfy the query.

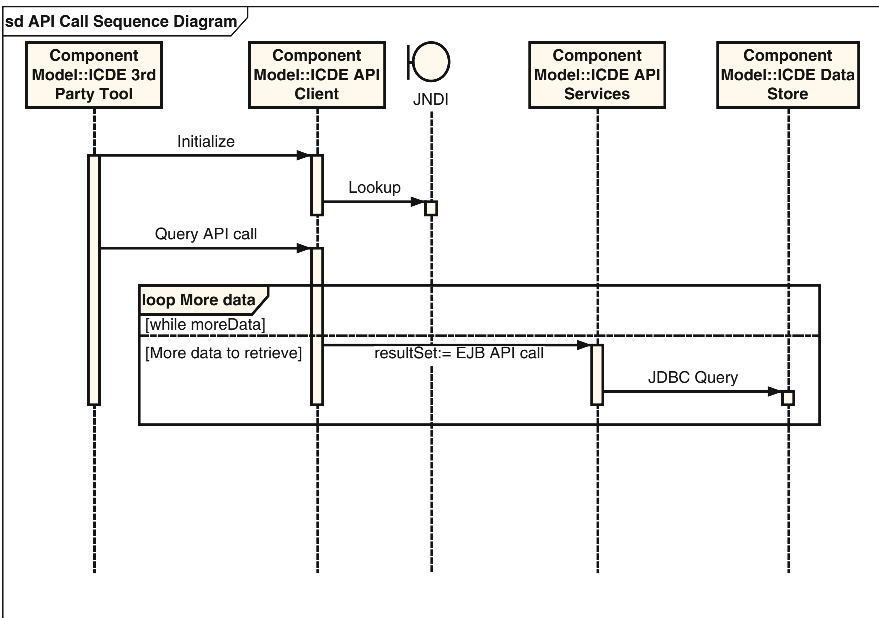

Fig. 9.5 Query API call sequence diagram

All the ICDE APIs that return collections of events may potentially retrieve large result sets from the database. This creates the potential for resource exhaustion in the JEE server, especially if multiple queries return large event collections simultaneously.

To alleviate this potential performance and reliability problem, the design employs:

- Stateless session beans that release the resources used by a query at the end of every call
- A variation of the page-by-page iterator pattern[4] to limit the amount of data each call to the session bean retrieves

The *ICDE API Client* passes the parameter values necessary for constructing the JDBC query, along with a *start index* and *page size* value. The page size value tells the session bean the maximum number of objects[5] to return from a single query invocation, and for the initial query call, the start index is set to NULL.

The JDBC call issued by the session bean exploits SQL features to return only the first *page size* rows that satisfy the query criteria. For example in SQL Server, the TOP operator can be used as follows:

```
SELECT TOP (PAGESIZE) * FROM KEYBOARDEVENTS WHERE (EVENTID > 0
AND USER = "JAN" AND APP_ID = "FIREFOX")
```

The result set retrieved by the query is returned from the session bean to the client. If the result set has *page size* elements, the *ICDE API Client* calls the EJB query method again, using the key of the last element of the returned result set as the *start index* parameter. This causes the session bean to reissue the same JDBC call, except with the modified *start index* value used. This retrieves the next *page size* rows (maximum) that satisfy the query.

The *ICDE API Client* continues to loop until all the rows that satisfy the request are retrieved. It then returns the aggregated event collection to its caller (the third party tool). Hence this scheme hides the complexity of retrieving potentially large result sets from the ICDE application programmer.

A sequence diagram depicting the behavior of a *write* API call is shown in Fig. 9.6. The write API call contains parameter values that allow the *ICDE API Client* to specify whether an event should be published after a successful write, and if so, on which topic the event should be published.

A sequence diagram for storing an ICDE user-generated event is shown in Fig. 9.7. An event type may require multiple JDBC INSERT statements to be executed to store the event data; hence the container transaction services should be used. After the event data is successfully stored in the database, if it is a publishable

---

[4]http://java.sun.com/developer/technicalArticles/JEE/JEEpatterns/

[5]The "page size" value can be tuned for each type of event to attempt to maximize server and network performance. A typical value is 1,000.

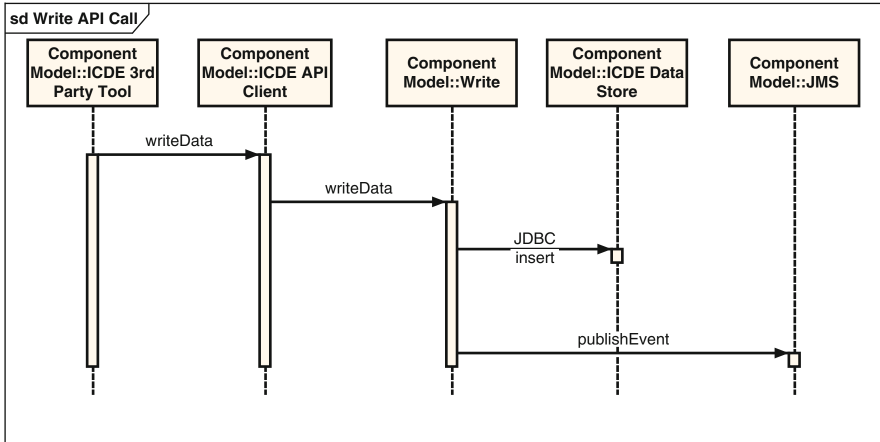

Fig. 9.6 Sequence diagram for the write API

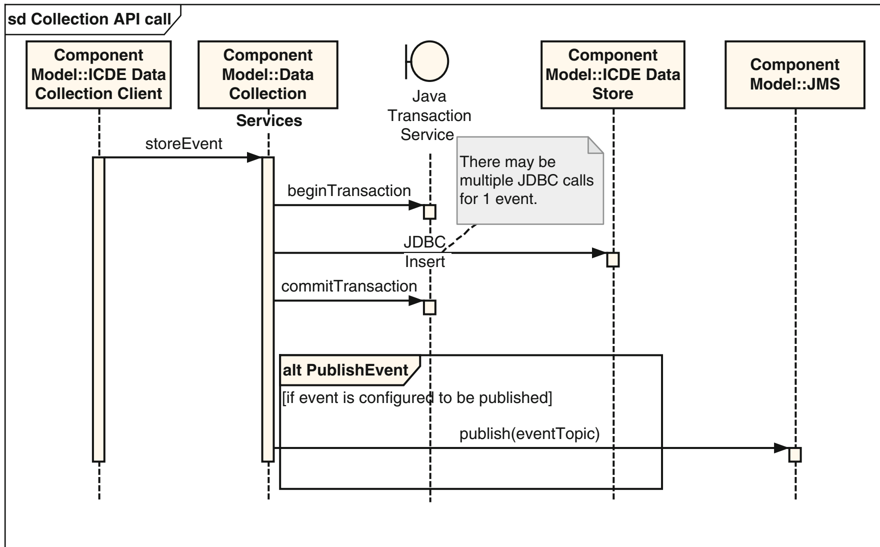

Fig. 9.7 Sequence diagram for storing user generated events

event type, the event data is published using the JMS. The JMS publish operation is outside the transaction boundary to avoid the overheads of a two-phase commit.[6]

[6]There's a performance trade-off here. As the JMS publish operation is outside the transaction boundary, there can be failures that result in data being inserted into the data store, but with no associated JMS message being sent. In the ICDE context, this is undesirable, but will not cause serious problems for client applications. Given the likely frequency of such failures happening (i.e., not very often), this is a trade-off that is sensible for this application.

9.4.5 Implementation Issues

The Java 2 Enterprise Edition platform has been selected to implement the ICDE v2.0 system. Java is platform neutral, satisfying the requirement for platform heterogeneity. There are also quality open source versions available for low-cost deployment, as well as high performance commercial alternatives that may be preferred by some clients for larger mission-critical sites. In addition, JEE has inherent support for distributed component-based systems, publish–subscribe event notification and database access.

Additional implementation issues to consider are:

- *Threading*: The *ICDE API Client* component should be thread-safe. This will enable tool developers to safely spawn multiple application threads and issue concurrent API calls.
- *Security*: ICDE tools authenticate with a user name and password. The API supports a *login* function, which validates the user/password combination against the credentials in the ICDE data store, and allows access to a specified set of ICDE user data. This is the same mechanism used in v1.0.
- *EJBs*: The *Data Collection Services* session beans issue direct JDBC calls to access the database. This is because the JDBC calls already exist in the two-tier ICDE v1.0, and hence using these directly in the EJBs makes the refactoring exercise less costly.

9.5 Architecture Analysis

The following sections provide an analysis of the ICDE architecture in terms of scenarios and risks.

9.5.1 Scenario Analysis

The following scenarios are considered:

- *Modify ICDE Data Store organization*: Changes to the database organization will necessitate code changes in the EJB server-side components. Structural changes that do not add new data attributes are contained totally within these components and do not propagate to the ICDE API. Modifications that add new data items will require interface changes in server-side components, and this will be reflected in the API. Interface versioning and method deprecation can be used to control how these interface changes affect client components.
- *Move the ICDE architecture to another JEE supplier*: As long as the ICDE application is coded to the JEE standards, and doesn't use any vendors extension

classes, industry experience shows that JEE applications are portable from one application server to another with small amounts of effort (e.g., less than a week). Difficulties are usually encountered in the areas of product configuration and application-server specific deployment descriptor options.

- *Scale a deployment to 150 users*: This will require careful capacity planning[7] based on the specification of the available hardware and networks. The JEE server tier can be replicated and clustered easily due to the use of stateless session beans. It is likely that a more powerful database server will be needed for 150 users. It should also be feasible to partition the ICDE data store across two physical databases.

9.5.2 *Risks*

The following risks should be addressed as the ICDE project progresses.

Risk	Mitigation strategy
Capacity planning for a large site will be complex and costly	We will carry out performance and load testing once the basic application server environment is in place. This will provide concrete performance figures that can guide capacity planning for ICDE sites
The API will not meet emerging third party tool supplier needs	The API will be released as soon as an initial version is complete for tool vendors to gain experience with. This will allow us to obtain early feedback and adapt/extend the design if/when needed

9.6 Summary

This chapter has described and documented some of the design decisions taken in the ICDE application. The aim has been to convey the thinking and analysis that is necessary to design such an architecture, and demonstrate the level of design documentation that should suffice in many projects.

Note that some of the finer details of the design are necessarily glossed over due to the space constraints of this forum. But the ICDE example is representative of a medium complexity application, and hence provides an excellent exemplar of the work of a software architect.

[7] Capacity planning involves figuring out how much hardware and software is needed to support a specific ICDE installation, based on the number of concurrent users, network speeds and available hardware.