Detailed specifications

Contents

1 Multiple clients
2 Transactions
3 Performance evaluation
    3.1 Average end-to-end delay
    3.2 Transaction abort rate
    3.3 Distributed evaluation
    3.4 Performance evaluation report
4 Suggested development plan
    4.1 Design considerations
5 Deliverables

Overview

You must extend your server to support multiple simultaneous connections from clients and support a simple transaction semantics.

Multiple clients

The skeleton storage server given to you only supports one connection from a client at a time. When a client has established a connection with the server, any additional clients that try to connect to the server must wait for the first client to disconnect from the server. Examine the server code and understand why this happens.

For this milestone, you must modify your server so that multiple clients can have open connections to the server at the same time and issue get, set, and query calls. The server should not starve any client and the clients should not be aware of one another. The server should support at least 10 simultaneous connections from clients.

There are different methods to support concurrency. This assignment is focused on using **threads** for concurrent processing of client requests at the server:

- When a new connection is made by a client, the server creates a new **thread** to handle the connection including receiving commands from the client and sending responses back to it. The thread ends when the client closes the connection.

However, you can earn some bonus marks if you support concurrency using any or both of the approaches described below in addition to the mandatory implementation mentioned above:

- The server maintains multiple open connections and uses the `select` system call to simultaneously listen for new connections and wait for data on existing open connections.

- When a new connection is made by a client, the server creates a new **process** to handle the connection including receiving commands from the client and sending responses back to it. The process ends when the client closes the connection.

Each alternative above has its own advantages and disadvantages. For example, in the *select-based* approach, the server only has one unit of execution, whereas in the other two approaches the server runs multiple units of execution concurrently which is potentially more efficient.

On the other hand, with threads and processes you must be careful that you *synchronize* access to any data that is shared among the threads or processes; you do not want two processes writing to the same data at the same time. Using `select` has its own complexities, however, since your code must be able to receive commands from any client at any time, while with a process per connection model, each process communicates with a single client similar to the skeleton server.

Think carefully about the concurrency model and what the implications will be for the `storage_get`,`storage_set`, and `storage_query` calls. Also think about how logging will be affected by concurrency, especially in the case where logging is output to a file.

Your server must support a new `concurrency` configuration parameter to indicate whether (and how) the server can handle multiple clients concurrently. The parameter value is an integer with the following values:

- 0: The server should not support multiple clients concurrently. Concurrent clients must wait for the currently connected client to disconnect. This is essentially the mechanism implemented in the skeleton server.
- 1: The server supports multiple concurrent clients using multiple threads.
- 2: The server supports multiple concurrent clients using one of the two optional approaches (`select` or `processes`).
- 3: The server supports multiple concurrent clients using the other optional approach from when the value is set to 2.

Here are some examples of the `concurrency` parameters.

```
concurrency 0
concurrency 1
```

You only need to implement the `threads` concurrency approach, so your server only needs to support `concurrency` values of 0 and 1. However, you can choose to implement the other approaches for bonus marks. Note that if you choose to implement these alternative approaches, you must also evaluate them.

Transactions

Consider a scenario where a client would like to update the population of Toronto in your server's census table (note *multiple* "columns are used for this example"). The client would like to keep the other columns in the record unchanged, so the client performs a `storage_get` to retrieve the Toronto record, updates the population in the record, then does a `storage_set` to store the record with the updated population count on the server:

```
// Retrieve the record
struct storage_record r;
storage_get("census", "Toronto", &r, conn);

// Update only the population column in r.value.
// ...

// Store the record.
storage_set("census", "Toronto", &r, conn);
```

In the case where multiple clients can connect to the server, it's possible that between the `storage_get` and `storage_set` above, another client updates the Toronto record, perhaps changing the rank column's value. Then when the first client performs its `storage_set` with the new population value, it will overwrite the new rank value set by the second client.

Transactions are a way to ensure that a sequence of operations are logically executed together as a unit and can be used to guard against the above scenario. For this milestone, you must implement a simple transaction scheme that allows a pair of `storage_get` and `storage_set` calls on the same record to be executed within a transactional context. In this way, if the record is updated by another client in the duration between the get and set calls, the set call will *abort*, that is, the record will not be updated.

Transactions should be supported as follows:

- The server associates a counter with each record. Every time the record is updated, the counter is incremented.

- When a `storage_get` is called, the record's counter is stored in the `metadata` field of the record structure that is passed back to the caller.

- To indicate that a `storage_set` is to be executed as part of the transactional context of the`storage_get`, the client application will call `storage_set` with the same record `metadata` field as returned in the associated `storage_get` call.

- Before updating the record as indicated in `storage_set`, the server compares the record's counter with the counter in the `storage_set`'s record `metadata`. If the counters match, then there have been no intervening updates to the record, and the updated record value is stored. Otherwise, the record is not updated, the transactions aborts, and `storage_set` returns -1 and sets errno to`ERR_TRANSACTION_ABORT`.

You can make the following assumptions about how the client application will call the `storage_set` function:

- If a `storage_set` is to be executed within the transactional context of a previous `storage_get`, the metadata field in the storage_set's record will be the same as that in the associated storage_get.

- If a storage_set is NOT to be executed within the transactional context of another storage_get, the metadata field in the storage_set's record will be 0.

Here is an example of how a client application performs a transaction.

```
// Get the record.
struct storage_record r;
storage_get("census", "Toronto", &r, conn);

// Update the population.  Do not change r.metadata.
strcpy(r.value, "Province Ontario, Population 2000000, Change 9, Rank 1");
int status = storage_set("census", "Toronto", &r, conn);
if (status == -1 && errno == ERR_TRANSACTION_ABORT) {
        printf("Transaction aborted.\n");
}
```

Here are the same operations called without a transaction.

```
// Get the record.
struct storage_record r;
storage_get("census", "Toronto", &r, conn);

// Update the population.
memset(r.metadata, 0, sizeof(r.metadata));  // Indicate a separate transaction.
strcpy(r.value, "Province Ontario, Population 2000000, Change 9, Rank 1");
storage_set("census", "Toronto", &r, conn);
```

You may not make any assumptions about the metadata field in the storage_get call.

The storage_set function should set errno to ERR_TRANSACTION_ABORT if the transaction aborts. TheERR_TRANSACTION_ABORT constant is defined in storage.h.

Notice that the transactional context as described above only supports a sequence of a storage_get call followed by a storage_set call. Notably, you do not need to support transactions that include an arbitrary number of gets and sets.

The above scheme relies on the client application not tampering with the record metadata, and hence is vulnerable to malicious client applications (i.e., also applications that incorrectly use the transaction abstraction we defined above). Think about what kind of attacks your server is susceptible to, and how you might guard against these attacks. You do not have to implement these security measures, but you should be aware of the possible attacks, and discuss their implications in the design document.

There is also a subtle issue with how you initialize the counter for a record. You should not simply set the counter for a new record to 0. Think about what would happen in the case where a record is deleted and inserted again within the transaction window of another client.

Think about how you could guarantee that the intended effect of multiple concurrent transactions is performed in spite of some of them aborting.

Performance evaluation

You will need to quantitatively evaluate how multiple clients affect the average end-to-end delay and how they affect the transaction abort rate.

Average end-to-end delay

Devise an experiment to measure the *average end-to-end delay* of a storage operation as the number of clients increases. You may evaluate any of the get, set, or query operations (or any combination of them). Compare the performance when the server supports multiple connections and when it does not.

For example, you can write a client application to repeatedly call `storage_get`. Measure the average end-to-end delay when there is only one client running. What happens to the average delay when there are two clients simultaneously issuing gets? Repeat this for up to at least ten clients.

You can use the same census workload and end-to-end metric from the previous milestones. **However, you will need to extend the** *'census table to retrieve multiple columns (*i.e: Province, Population, Change, and Rank*) to support* `storage_query`'. You should compare at least two server configurations: one that does not support multiple clients (`concurrency 0` in the configuration file) and one that supports the mandatory concurrency mode (`concurrency 1` in the configuration file). If you implement more concurrency mechanisms you should also evaluate them in order to receive the bonus marks. You can also evaluate using a different logging parameter.

The results should be presented on a chart where you vary the number of clients on the independent (horizontal) axis, and plot the average end-to-end delay on the dependent (vertical) axis. There should be one data series in the chart for each server configuration you evaluate.

Transaction abort rate

The likelihood of a transaction aborting increases with the number of clients. Run another set of experiments to measure how the number of clients effects the *transaction abort rate*. The transaction abort rate is the ratio of aborted transactions to the total number of transactions.

You can use the same census table as above, but your clients must now repeatedly issue get and set operations within a transactional context. You must devise a way to make sure that some transactions will abort. One way to achieve this is to randomly perform the gets and sets on a small number of records and run the experiments long enough to allow for some transactions to abort. You can also add a small delay at the client application between a get and set call to artificially increase the transaction window and increase the chance that the record is updated by another client within this window and cause a transaction abort. You can also enable logging to file to increase processing time.

Similar to the above experiments, the results should be presented on a chart that plots how the transaction abort rate varies with the number of clients. Again, there should be one data series in the chart for each server configuration, and you should evaluate at least the mandatory concurrency configuration.

Distributed evaluation

In both sets of experiments above you will be running several processes: at least one process for the server, and one process for each client. To more accurately represent a situation where there are multiple users of your storage server, you should run each client on a **separate machine**. Doing this also ensures that your results are not influenced by an overloaded CPU when so many clients are run on the same machine.

Be careful about how you setup your experiment. For example, if you want to evaluate the average end-to-end delay with two clients running on separate machines, it is very likely that one client will start running slightly before the other, and the delays of the first few operations by the first client (and the last few by the second client) will not occur at the same time as the other client's operations. One way to deal with this is to discard the results of the first and last few operations and only use the delays of those operations you're sure are issued concurrently with the other clients.

While there are dozens of lab machines available for you to use, do not wait until the last minute to run your experiments or it might be difficult to find an unused machine. You can use the `ps faxu | less` and `top` commands to see if there are others using the machine.

Experiments on shared resources such as the lab machines can be unpredictable since they are affected by factors beyond your control such as the network traffic and CPU usage by other users. For this reason, you should run your experiments several times and average the results you get from multiple runs. You should convey the **distribution of the results** across these runs, using techniques such as error bars to show the standard deviation or confidence intervals.

Conducting experiments in a distributed environment is not easy and will take longer than you think. Do not wait until you have finished the implementation before you start the performance evaluation. Since one of the server configurations requires no additional coding (i.e., the server with no concurrency), one of your team members can start getting results for this configuration right away. This gives you a chance to resolve any evaluation issues early on and make the evaluation of the other server configurations much smoother.

Performance evaluation report

You must write a report that describes your evaluation *methodology*, *results*, and *analysis*. This will be handed in twice: once with the code to be graded by TAs, and once to Turnitin.com to be graded by your CI when you submit your code.

The methodology section should be a brief description of the evaluation setup. This includes

- the concurrency mechanisms you evaluate
- the metrics you measure
- the workload including the tables used, and the operations
- the distributed setup including the number of concurrent clients you evaluate

The results section should present the quantitative values from running the experiments. Unlike in the previous milestone, this time you must present the results graphically in a chart including a measure of the distribution of the results across multiple runs.

Finally, in the analysis section, you should draw some conclusions you derive from the evaluation results. Summarize the performance of your server under different configurations with multiple clients. Do you see an overhead with supporting multiple clients? Does it seem like your server can only handle a certain number of clients? Does the transaction abort rate increase rapidly beyond a certain number of clients?

The evaluation report should contain no more than 500 words (roughly one page), excluding graphical material.

Suggested development plan

Here are some of the major tasks involved in this assignment.

- Overal design given the above specified extensions
- Implementation of concurrency
  - Thread management (initialization, pools, cleanup, etc.)
  - Select-based concurrency - optional
  - Process-based concurrency - optional
- Data synchronization
- Implementation of transactions
  - Adding counters to records
  - Setting and checking record counters
- Changes to configuration file
- Performance evaluation and analysis
- Design document

Again, it is up to you whether you want to divide the tasks as above.

Design considerations

You need to summarize key design decisions you will have to make when designing your storage server. These design decisions are:

- What are the implications of using threads as a concurrency mechanism?
- How will access to shared data be synchronized (e.g., do you need to use a form of locking)?
- How do you protect your log files from concurrent access and modification by multiple threads?
- How are threads managed? How are they created and destroyed?
- In what ways can a malicious client application tamper with the record metadata and effect transaction processing? Does your server defend against these attacks?

You should argue about why you came to your design decisions and discuss the pros and cons of the decisions, such as the ease of implementation, robustness to failures, performance implications, and so on. Note, for many decisions you will have to balance trade-offs. Try to understand what they are.

Deliverables

As usual, there are two deadlines for this assignment. We first ask you to hand in your design documents, later followed by the code you developed.

By the design document deadline, you must hand in the **M4 Design Document: Scalable and Concurrent Server** (M4 DD, rubric here), which contains the following content:

- *Executive Summary* revised to reflect suggestions made by your Communication Instructor/Project Manager and to reflect new design decisions.
- *M3 Design Document* revised, edited, updated and proofread implementing any suggestions or corrections suggested by your Communication Instructor/Project Manager and reflecting any new design decisions.
- *New Design Decisions.* Major design decisions are discussed above in Design Considerations.
- *Lessons Learned.* In addition to the M4 Design Document, each team member must write a 500-600 word personal statement. This statement should discuss challenges and/or learning opportunities afforded by the course in **the following three areas**: 1) programming, 2) written **or** oral communications, and 3) team work. For this statement, you may want to ask yourself, "Knowing what I know now, what would I do differently if I could start this course again?" "How would I handle situations differently?" Thus, we ask you not only to assess your performance in the course but also to propose strategies for improvement. This statement should include an introductory paragraph, stating the main purpose of the piece; body paragraphs, presenting and supporting each of your main points; and a conclusion, concisely summing up your analysis. Each student must submit this statement separately to Turnitin.com (not as a part of the M4 DD submission) by the M4 DD deadline (March 30).
- *Performance Evaluation Report:* This report is due with your M4 code (April 6). The Performance Evaluation instructions are stated above in this file.

Total pages of text, not including *Cover Page, Executive Summary, Table of Contents*, *List of References, Appendices, Lessons Learned,* and *Performance Evaluation,* must be no more than thirteen (13) pages. For this document, you may include up to six (6) pages in the appendices. Please refer to the grading rubric for further information.

The text must be submitted to Turnitin.com. For information on registering with Turnitin.com, please view Turnitin Registration Information here.

By the software development deadline, you must hand in your software artifacts that includes at least the following content.

- You must hand in your code.

- The *Performance Evaluation Report.* This should follow the performance plan and provide some analysis of the results. For example, were the results as expected, were they surprising, what do the results say about the design decisions you have made. This document must be submitted to your CI through Turnitin.com as well.

back to top