Reservation Server

Evaluation

digisoln.com

Table of Contents

Overview

A reservation system has been built for a restaurant for Christmas lunch that multiple customers can concurrently interact with. The following document outlines the instructions for this system, as well as an exhaustive listing of the results from the testing undertaken from the Design Documentation. Finally, this documentation will include a discussion of current bugs and limitations of the system.

Acceptance Documentation

Following a list of instructions, the completed acceptance testing and bug / limitation discussion can be used to best identify successes and areas of improvement of the reservation system.

User Instructions

The following instructions will be supplied to the user via email, web and text file with the client console:

1. Using your file browser, locate and double-click the file **ReservationClient.exe**
2. Windows Firewall may ask you to allow network access to this application – select Allow Exception
3. If you are connected properly, you will see a "Successful connection to server" message.
    a. If not connected, you may be behind a proxy server, your internet may be down or we may be experiencing a server outage. Make sure your router has TCP port 12345 unblocked.
4. With the user menu options in front of you, simply type the number of your selection and press enter:
    a. To request seating availability information, type "1" (no speech marks) and hit enter.
    b. To make a reservation, type "2". You will be asked for some details including name, sitting number and number of places required. Type this information and hit enter when complete. When the reservation is complete, you will receive a RESERVATION KEY that is necessary if you want to make changes to your booking. If a reservation cannot be complete (due to places) you will be informed.
    c. To cancel a reservation, type "3". You will need to supply your own unique RESERVATION KEY that was given to you when initially confirming your reservation.
5. Hit "q" or simply close the window at any point to end the reservation system.

Test Results

The following acceptance test results outline the risks and allowable benchmarks (where necessary) for each phase of development. Due to page size restrictions, the **testing condition has not been rewritten in this discussion** – however the *test result letter* corresponds with the *test condition letter* from the Design Documentation:

Phase	Test Condition Result (Indexed by letter – original test condition can be found in Design Documentation)
1	a. Booking *Hashtables* stores Reservation structures containing all necessary data as VALUE fields ☑ b. static integer unique_reservation_key increments by 1 for each booking taken ☑ c. Restaurant class contains full implementation of functionality as required ☑ d. Main restaurant_sitting array of sittings (containing Booking Hashtables) is made private ☑ e. Based on software requirements, information about sittings (e.g. sitting name labels) and the multiplicity of sittings (the assignment only required 2 but this was made selectable) could be considered redundant. Furthermore, the Assignment 1 task specifically states that "all is required is an association between sittings and seats" – which implies that customers cancel by number, not by name... although the reservation key (and storing customers name etc.) was not required, and could be considered redundant, it was included to significantly increase appeal of the prototype. ☒
2	a. Mutexes - Monitor::Enter(this) and finally {Monitor::Exit(this)} – prevent race conditions when adding and removing a booking within the reservation server. ☑ b. While (not enough places) Monitor::Wait(this) – condition variable waits for notification (::Pulse) from cancellation thread when booking to allow new reservations to wait for places available. ☑ c. Because of preceding *test condition results a* and *b*, seating cannot be over allocated. ☑

	d. Thread::Sleep(5000) for 5 seconds within thread request (and Sleep(1000) on all) as required ☑
3	a. Timeout / failed to respond error (throws SocketException so no freeze). It must be noted client will be distributed binary and thus will not be able to change server IP (or nameserver). Still, given the timeout, technically this could be seen as a test failure result. ☒ b. Similar to *result condition a*, using the wrong port will cause the server to refuse the connection. See bugs / limitations for further discussion here. ☒ c. Of course the system requires both client and server at minimum to make a booking, however the server listens if a client is not present, and a client will attempt to connect to a non-existent server on start-up only. Given this, enough functionality exists here to be deemed a pass. ☑ d. Both the client and server can disconnect at any point during the booking system – the client transactions will cease and the client console must be restarted once the server is rebooted. The data structure is lost if the server fails. This is a limitation (covered below), clearly not ideal, but for the requirements of the application, again enough functionality exists to not fail this test. ☑ e. Thread ^thr is parameterized and dynamically allocated the address of each incoming TcpClient ☑
4	a. Messages repeated in server console – clearly no anomalies occur when processing messages ☑ b. Client thread method in TcpServer class dynamically allocates memory for booking / cancelling threads –these are fully qualified operating as one "system" – using mutexes, notification, etc. ☑ c. Isolation – passed (mutex's) ☑, consistency – fail (allows erroneous data) ☒, durability – fail (transient data structure) ☒, *atomicity* – some evidence of pass (isolation and distribution of client console aid atomicity). More atomicity testing required (outside the scope of this project). ☐ d. Feedback system while adding booking demonstrates what could be done with cancellation. ☑
5	a. No deadlock at time of publish. read_unknown_length() until next input is robust until EOM. ☑ b. Software functionality specifications exceeded. ☑ c. Use of Stream reader and writer is logical abstraction of "stream". String parsing methods useful ☑

Bugs / Limitations

The system in its current state exceeds the required specifications of the project. However, at an industry standard the following bugs and limitations must be addressed. Initially, the biggest short coming is the lack of a GUI (specifically on the client side) and the lack of user proofing – erroneous and syntactically incorrect entries lead to crashes on the server side and frozen states on the client side.

Testing turned up one significant limitation with the current state of thread management. From server perspective, assume a full house with two incoming new reservations – thread "new1" and thread "new2". Neither allowed initially - both are waiting for cancellation pulse (condition variable using ::Wait in addBooking method). Thread "new1" sent in first (oldest) – then cancellation is sent in with enough seats for either one (but only one) of the new threads – then "new2" reservation request sent in last (newest). The oldest thread appears to be placed on the *bottom of a system stack* with the newest request on top – in reality the ::pulse may simply be notifying the newest (last) thread in. Regardless, the situation may be better served if waiting threads (in this instance) were FIFO queued (first customer in should have first choice). Thus, control over the current (seemingly) LIFO stack handling of waiting threads by the system may need further investigation. Furthermore, this could be solved by investigating *thread priorities*, and implemented in the code where timeouts are measured in makeBooking method (t>10000, t>20000, etc). It is warranted that these threads close to the end of their lives may be marked with increasing priority.

Currently Thread::Sleep(x) to model processing times is far from industry standard but included *inside* the critical sections of adding and cancelling a booking to best model data access and processing times.

In terms of network programming, the server for real-world implementation requires a dedicated server IP address, as 127.0.0.1 currently used is only a loopback IP on a single machine for testing. An unused private port must also be chosen and registered with http://www.iana.org. Currently TCP port 12345 is used by Trend Micro's Office Scan

products, Windows Mobile App “Ambient Color”, NetBus Trojan Horse, as well as many other trojan horse and backdoors – which is a tremendous compromise to network security.

Feedback for the cancellation thread can be modelled upon the feedback mechanism for the add booking thread. Currently, no timeout messages or incorrect key feedback exists for cancelling a reservation.

Currently, no administrator backend, or reports on reservations, nor any other required functionality for a real-world client restaurant exists. Off-site data file backup is also necessary – the data storage is currently transiently stored. To allow this data autonomy, and to increase data independence, a DBMS should be used to avoid difficulty in compatibility with future legacy application changes.

Server instructions do not exist at the time of publishing (this was not a requirement of the software). Note to run the server application, double-click the **ReservationServer.exe** file.

Future Directions

The system has been implemented to a prototype standard. The above limitations must be addressed in order to make this system commercially viable.

References

Unless otherwise specified, this assignment is entirely my own work, utilising only these references:

Fraser, S. 2006. Pro Visual C++/CLI and the .NET 2.0 Platform. Berkeley, CA.

Jarvis, J. and Jarvis. D. 2009. Application Development using Visual C++ 2005.