Software Components
COMP 4002

Assignment 1

Semester 1 2016

Distributed Brute Force Password Recovery

Justin Kennedy
18070651
01/06/2016

Contents

Introduction

As part of the unit Software Components students were asked to implement a distributed brute force password recovery program. This report will explain the structure of the system and offer analysis of the design. The program was implemented in C# using .NET.

Requirements:

The program was to be implemented in .NET and make full use of distributed parallel processing. Additionally, it was required to:

- Have a GUI
- Implement callbacks to update the GUI
- Have a web services component

The program was to generate a random password of a given length and then proceed to recover the password in plain text form.

System Overview

The system was split into three tiers:

1. The presentation tier, consisting of the windows WPF client and the web page
2. The management (business tier), consisting of the server that deals with the logic of managing the services
3. The service tier, consisting of the machines that implement the brute force recovery.

The tier separation between the management and services is somewhat arbitrary because they could be considered one tier. However, since their concerns are very different, the tier separation makes sense.

In this system, the manager is the single point of contact for all of the machines in the system. Therefore, the manager updates both the GUI and web page, and directly manages the all the recovery services.

Problem Description and Approach

Basic Approach

The passwords that can be generated were taken from a character pool containing digits (0-9), lower case letters (a-z) and capitols (A-Z) giving a total pool size of 62. When combined with the length of the password, the total possible passwords (permutations) that the password can be is $Length^{62}$. Therefore, one of the main requirements of the system is an intelligent way to split the complete list of permutations among several recovery services for distributed computation.

This was achieved by implementing a batch system, whereby services would be assigned a batch of permutations to test against the encrypted password. A batch consisted of a range of permutations, such as 0 to 9000. Here, the permutation number means the number in the sequence of passwords if all permutations were generated by starting with a password containing only the first letter in the charset. The sequence continues by changing the last character until all possible combinations are tried, after which the second last character is changed and so on. For example, if the char set was ‘A’, ‘B’, ‘C’, then for a 3 letter password, the permutation sequence and permutations numbers would be:

Permutation	**Permutation Number**
AAA	0
AAB	1
AAC	2

ABA	3
…	…
CCC	26

Conveniently, the process of converting from the number to permutation is quite straight forward. This is because each decimal permutation number represents the permutation in n-ary form. Thus to convert from the number to the permutation, an n-ary conversion is required. This algorithm can be seen in the convertToPermutation(long permNumber) found in the RecoveryServiceImpl class.

Since the conversion can be completed in this way, it frees up the manager server from having to generate each of the permutations itself before passing them to the recovery services. Instead, the manager can simply pass a pair of permutation numbers to the services who can convert each number in the range to a permutation string for testing. Therefore, all the manager has to do is decide a batch starting point and ending point, which can be done in very trivially (and therefore quickly). The advantage here is that services have to wait very little time for the manager to generate the batches (it is an O(1) operation – far superior to O(n) for n number of passwords). Additionally, the size of the batch data is kept very small since there is no passing of permutation lists as parameters. Finally, the batch generation is not created recursively, so there's no possibility of a stack over flow exception. Thus, extremely long passwords can be handled by the batch generation process with ease.

In terms of design, using the batch system created a very flexible code. Firstly, a batch could be defined relatively easily. Secondly, the batch size could be made to be any length. This meant that the batch size could be calibrated to both the size of the job (total possible passwords) and the number of services. Thus only simple decision logic is required. For instance, when a job is small, the batch size can be an even split among the number of services. Alternatively, if the job is large, then the batch can be capped so that services don't take too long on a job. This avoids the situation where a service spends hours on a batch and encounters an error and all the hours of work is lost. With frequent returning of smaller batches, the work is "saved" and errors do not cost the job more time.

Implementation Description

Interfaces

A total of 5 interfaces were created in this system. The manager implements two interfaces:

1. IRecoveryManagerService: This describes how a service can contact the manager.
2. IRecoveryManagerGUI: This describes how the GUI can contact the manager.

In addition, there are two callback interfaces relating to the above interfaces:

1. IRecoveryManagerServiceCallback. This is implemented by the recovery services, and describe how the manager can contact the services
2. IRecoveryManagerGUICallback. This is implemented by the GUI, and describes how the manager can contact the GUI.

The fifth interface is the IRecoveryManager interface, which is essentially empty and inherits from both the IRecoveryManagerService and IRecoveryManagerGUI.

Splitting the interfaces in this way ensures that the manager exposes only the appropriate functionality to other parts of the system. For example, the GUI does not need to know about functions relating to batches, and the services doesn't need to know about functions relating to updating the GUI. This increases improves the maintainability and extendibility because functions are clearly related to only one part of the system.

Similarly, the dual use of callback interfaces ensure that the manager can only call functions on remote objects appropriately. Additionally, since the callback interfaces are split, modifications to the presentation tier won't require a recompilation on the service tier, and vice versa.

The alternative to this design was to have one callback interface inherit and specialize another. This would mean that care would need to be taken to ensure functions are called appropriately. It would also mean that changes to parent interface would require recompilation of all tiers, which could mean many thousands of reinstallations.

Class Overview

RecoveryServiceImpl

This class represents the manager server object. Its job is to assign batches to the services and manage their requests. It is also the point of contact for the GUI and web services.

PWRecoveryJob

This is a class internal to RecoveryServiceImpl and as such is only used by it. The purpose of this class is to manage the batch creation and to keep track of the status of the job (eg in progress, password found etc).

PWBatch

This class represents a single batch and is used by RecoveryServiceImpl and PWRecoveryJob. Its purpose is to wrap the Batch struct and store house-keeping data such as the time it was created and the service it was assigned to. This object is distinct from the Batch struct because it is more of a housekeeping class relating to the management of batches, whereas the struct is the raw data relating to the batch. This means the struct can be sent across the network and that the services don't need to know about, or receive the housekeeping data.

RecoveryServiceImpl

This is the recovery service class, which does the basic computation of generating and encrypting each password in a batch.

MainWindow

This is the GUI class, and handles all user input and output.

Web Services

The page is rendered using an .aspx web form. The calls to the system are made using ajax calls. This allowed for an asynchronous update to the web page when the password is found.

Analysis: Design

Extendibility and Maintainability

The design of the system tends to favour extendibility and maintainability over optimization (even though both were given due consideration). For example, the RPC call that assigns the batches to the services includes the char set every time, which increases network traffic overhead. The char set could have been hard coded to the services since it is final in the requirements. That does not represent the most extendable design. Instead, the charset was sent with each batch so that in future versions, the services could conceivably handle a wider variety of jobs. That would require very little recoding of the services. Additionally, the services were designed with a service-oriented approach: the services were to

offer the services of completing a batch of a given definition. It is not their job to know anything else, such as hard-coded specifics of the other tiers.

The sending of the char set with each batch was chosen purposefully to allow flexible choice of char sets. Although this was not implemented (yet), the user could choose to reduce the character pool to just letters. The recovery services would have no problem handling this at all, and the services would be able to recover the password in a much shorter time.

As mentioned, the downside to this implementation is that more data is sent across the network. This could be mitigated by setting the char set once at the start of a job. This has not yet been implemented.

Security

This system is far from battle-hardened. If someone were to connect to the manager server and send bogus data back, such as alerting that a batch is complete when it hasn't, the system could fail to find the password. Additionally, the recovered password is sent back to the manager and GUI in plain text, which means the recovery of sensitive passwords need to be done on secure networks only.

Parallel Processing Optimization

As mentioned before, a distinct advantage of the system is that the manager does not generate any of the passwords. An O(1) batch generation process is done in lieu of an O(n) generation. This means that all processing is able to be performed in a fully distributed manner. However, the way that the services generate the passwords for testing is not optimal. Currently, they convert each permutation number to the password string. This means a lot of the computation is repeated. For example, generating the password "AAAAAAAAAA" and "AAAAAAAAAB" requires only one character change, but the services recomputes the entire string from scratch. This is not the most optimal way of doing it, and therefore represents a weakness of the system. However, one could argue this is not a design flaw (but rather an implementation flaw) because there is nothing the design that forbids implementing the permutation generation in the optimal way. This optimization has been left out purely due to time constraints.

Another optimization weakness is that each batch is completed using only a single thread. This is far from optimal, since a better design would split the batch into two or four (depending on whether it is a dual core or quad core processor) and proceed from there. This weakness is not a significant weakness, since the user can simply open two or more instances of the recovery service to get full thread optimization.

Scalability

Without the tools to properly test the system (ie a hundred thousand computers) it is hard to determine what sort of behavior would emerge from the system. However, it is certain that the manager server would become the bottleneck to the system. The way that the manager is designed requires that many sections of the code are executed by one thread only. This can be seen in the code as lock(servicesLock){…}. The locking of the code in this way is required because the house-keeping data needs to be accessed synchronously. This data includes the list of services, the idle services queue and the list of assigned batches. Care has been taken to lock the minimum amount of code. Particularly, any RPC is *not* found in the locked code because this can potentially take a long time and therefore slow the manager server down considerably.

This synchronization of execution will slow the recovery process down when the system is scaled to hundreds of thousands or millions of machines because the manager could potentially receive millions

of requests, all of which require some degree of synchronization. This would cause an enormous queue to develop.

Another scalability issue is the choosing of the batch size. Currently, the batch size is capped at 150000 passwords. This number was chosen purely to show-case the systems batch-oriented design. The batches take roughly 10 seconds to complete in the labs. This is not intended to be an optimal number for large numbers of machines, but it works fairly well for small passwords and a few machines.

The cap on batch size can be changed to suit the machines and environment the system is deployed. This can be achieved by changing the hard coded values *estimatedRate* and *targetBatchTimeSec* in PWRecoveryJob, or it could be achieved dynamically by calling *setEstimatedRate(…)* and *setTargetBatchTime(..)*. If the system is deployed with hundreds of thousands of machines and keeping the default batch size, it is possible that it would take longer than 10 seconds for the manager to assign each service a new batch as they complete them. This would cause a queue to develop, and the system would not be making full use of the parallel processing available. This situation illustrates that the system is significantly sensitive to the value chosen for the maximum batch size. Too small, and the services queue up for new batches and network overhead becomes significant. Too big, and the services will risk losing hours of work if they fail. Additionally, setting the batch to a large number doesn't guarantee optimization, because a million services could still complete large batches all at the same time. One potential solution to this is to sleep the services a random amount when a timeout occurs so that their completions are staggered. (This solution has not been implemented - yet.)

Robustness

The generation of batches is performed equally well for small or large passwords, so in this sense, the system demonstrates a degree of robustness. In addition to this, the capacity of the system to handle varying char sets is also a form of robustness.

Another example of robustness is the cleanup process, whereby when all batches have been assigned (but not all have completed), a thread is spun to loop through the list of assigned batches and test their age against a maximum time. The oldest batch is chosen and if it has taken too long, it is assumed to be assigned to a service that has disconnected, and therefore is given to another idle service to complete. If the oldest batch is still young enough to be completed, the cleanup process sleeps until the point at which it becomes old enough to be checked again.

One weakness of the system is that it does not respond to the situation where all services have disconnected. This is because the manager does not implement a loop that polls to check the services. Instead it is an event oriented system whereby a service is only given a new batch upon returning a completed one. This means, beyond the initial assignments that kick start the process, the assignment of a new batch is triggered rather than polled. The aim of this approach was to reduce wasted processing (polling), but it makes the system blind to the possibility of having no connected services.

User Interface: usability

The GUI was designed to provide a high degree of usability and feedback. In a system like this, the user needs feedback in order to make best use of the program. Thus the user is given important information (every second) about the number of passwords tested, the rate at which they are tested, the number of connected services, the elapsed time and the percentage of passwords completed (see figure 1 below).

This allows the user to gauge the performance of the system and make decisions about when and how to use it.

A downside to the GUI is that the diagnostic data (rate, number tested etc) is dependent on the batch size. No diagnostic data will be seen until the first batch is completed, so if processes take several hours to complete a batch the user might think the system is not working. This problem is compounded with further updates possibly taking hours as well. However, with a target batch time of 10 seconds (for the smaller jobs this is not really an issue.

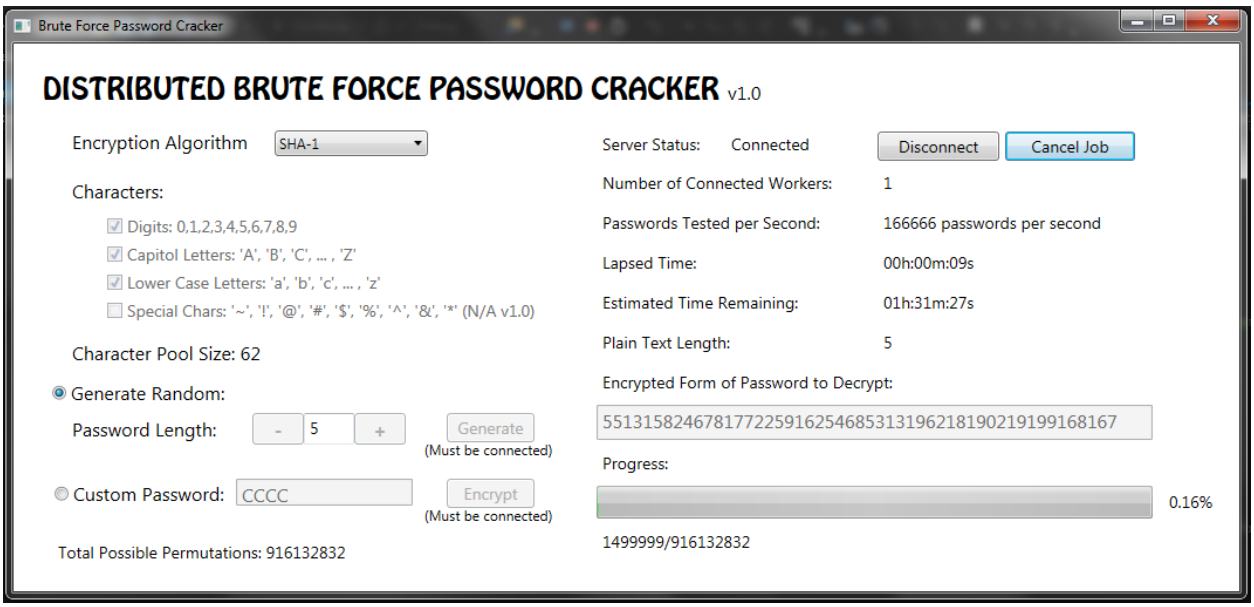

Figure 1: Windows GUI for the system

The web form diverges from the usability based design and is extremely bare and simplistic. It is more a proof of concept for the behind the scenes logic for accessing via web. It is definitely not a feature of the system and could be improved with better feedback from the manager server.

Analysis Conclusion

The system was designed with a tendency to favour maintainability and extendibility over optimization. As such the system does use more network traffic than is necessary with the batch system implemented. While there is some areas that the system could be further optimized, the job of recovering the process is split fairly well across the machines with a time complexity of O(n/c) for n number of passwords and c number of services connected. The system strikes a good balance between good modular / extendible code and processing optimization.