

**DEPARTMENT OF COMPUTER SCIENCE**  
**ASSESSMENT DESCRIPTION 2016/17**  
**(EXAM TESTS WORTH ≤15% AND COURSEWORK)**

**MODULE DETAILS:**

Module Number:	6	Semester:	1 and 2
Module Title:	Distributed Systems Programming		
Lecturer:	Dr David Chalupa / Dr Nina Dethlefs		

**COURSEWORK DETAILS:**

Assessment Number:	1	of	1
Title of Assessment:	CheeseCakeSoft Distributed Encryption Service		
Format:	Program		
Method of Working:	Individual		
Workload Guidance:	Typically, you should expect to spend between	40	and 60 hours on this assessment
Length of Submission:	This assessment should be <b>no</b> more than: (over length submissions <b>will be</b> penalised as per University policy)		N/A - coding exercise <b>words</b> (excluding diagrams, appendices, references, code)

**PUBLICATION:**

Date of issue:	9 <sup>th</sup> March 2017
----------------	----------------------------

**SUBMISSION:**

ONE copy of this assessment should be handed in via:	Canvas	If Other (state method)	
Time and date for submission:	<b>Time</b>	14:00	<b>Date</b> Thursday 11 <sup>th</sup> May 2017
If <b>multiple hand-ins</b> please provide details:			
Will submission be scanned via TurnitinUK?	No		

The assessment must be submitted **no later** than the time and date shown above, unless an extension has been authorised on a *Request for an Extension for an Assessment* form which is available from the Departmental Office (RB-308).

**MARKING:**

Marking will be by:	Student Name
---------------------	--------------

**COURSEWORK COVERSHEET:**

NO coversheet required.
-------------------------

**ASSESSMENT:**

The assessment is marked out of:	30	and is worth	50	% of the module marks
----------------------------------	----	--------------	----	-----------------------

**N.B** If multiple hand-ins please indicate the marks and % apportioned to each stage above (i.e. Stage 1 – 50, Stage 2 – 50). It is these marks that will be presented to the exam board.

**ASSESSMENT STRATEGY AND LEARNING OUTCOMES:**

The overall assessment strategy is designed to evaluate the student's achievement of the module learning outcomes, and is subdivided as follows:

LO	Learning Outcome	Method of Assessment {e.g. report, demo}
<b>2</b>	<i>Discuss and make judgements through critical analysis and evaluation in relation to the principal characteristics of distributed applications and their impact on design, implementation and deployment, justifying your arguments</i>	Demo
<b>3</b>	<i>Critically evaluate a range of contemporary distributed computing technologies, integrating reference to literature effectively with own ideas</i>	Demo
<b>4</b>	<i>Specify, design and implement a distributed software application, which is both appropriate and relevant for a suggested purpose</i>	Demo

Assessment Criteria	Contributes to Learning Outcome	Mark
Basic functionality (Hello message, sorting and array transmission, elementary communication, special case handling)	3,4	6
Hashing, encryption and digital signature implementation (public key handling, hashing, transmission of encrypted data, decryption, digital signature verification)	3,4	5
Use of concurrency (simple concurrency handling, multiple clients running, server handling multiple connections)	2,4	5
Robustness (dealing with random data, large amounts of data and very large numbers of clients)	2,3,4	4
Software design (general design, functionality, quality of code, maintainability, innovation)	2,3,4	10

## FEEDBACK

FEEDBACK			
Feedback will be given via:	Canvas	Feedback will be given via:	
Exemption (staff to explain why)			
Feedback will be provided no later than 4 'teaching weeks' after the submission date.			

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

You are advised to read the **NOTES** regarding late penalties, over-length assignments, unfair means and quality assurance in your student handbook. In particular, please be aware that:

- Your work will be awarded zero if submitted more than 7 days after the published deadline.
- The overlength penalty applies to your written report (which includes bullet points, and lists of text you have disguised as a table. It does not include contents page, graphs, data tables and appendices). Your mark will be awarded zero if you exceed the word count by more than 10%.

Please be reminded that you are responsible for reading the University Code of Practice on the use of Unfair means (<http://student.hull.ac.uk/handbook/academic/unfair.html>) and must understand that unfair means is defined as any conduct by a candidate which may gain an illegitimate advantage or benefit for him/herself or another which may create a disadvantage or loss for another. You must therefore be certain that the work you are submitting contains no section copied in whole or in part from any other source unless where explicitly acknowledged by means of proper citation. In addition, **please note** that if one student gives their solution to another student who submits it as their own work, **BOTH** students are breaking the unfair means regulations, and will be investigated.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility, not the Department's, to produce the assignment in question.

# CheeseCakeSoft Distributed Encryption Service

You have been hired as a distributed systems specialist by a security software company called *CheeseCakeSoft*. You were promised that your first task will not be too hard. And it turns out that it really is not. The task will be to develop an internal client / server application for the company, which provides a set of **extra secret** security and encryption services. Kind of...

At a meeting with the CTO, you were instructed that the client / server application must be able to do the following:

- Sending a **hello message** from the client to the server, indicating the identity of a client to the server. The server will have to be able to handle multiple client requests at any time. This relates to all requests, not just the hello request.
- A **sorting routine**, which a client can use to delegate its job of sorting some numerical data to the server. Because, well, sorting data is tedious.
- A routine allowing the client to ask the server to provide the **server public key** to the client. This public key will be specific to your server. This way, the server will indicate its identity to the client and they can communicate securely.
- Computing **SHA-1 hash** of a message on the server side and providing it in a hexadecimal form back to the client.
- Computing **SHA-256 hash** of a message on the server side and providing it in a hexadecimal form back to the client. Somebody told you that SHA-256 is more secure than SHA-1.
- Transmitting a message from the client to the server in an encrypted form by the **RSA cryptosystem**. That means:
  1. encrypting the message on the client side (using the public key that your client has to first request from the server);
  2. transmitting the message in an encrypted form to the server;
  3. decrypting the message on the server side (and hope that it is the same as the input on the client side).
- The server should also be able to **digitally sign** some data for the client using the private key of the server. You must use the RSA cryptosystem for this, combined with the SHA-1 hash function. However, no signing of a cheesecake is allowed, even if it is a CHEESECAKE! The client then has to be able to verify the server's signature using the public key requested from the server.

Your task is to follow the next instructions carefully and develop a **console-based** client / server application, which is able to provide the following functionality. Note that **each identifier (solution name, project names, all relevant project directories and all of the relevant code) in your solution must conform to these instructions**. Otherwise, your application may not pass the automatic unit tests, because the testing software will simply not find your files at the place, in which they are expected.

## Instructions

Your task is to create a Visual Studio solution called **ACW\_08346\_abcdef**, where **abcdef** has to be substituted by **your actual 6-digit Campus id**. For example, if your Campus id is 123456, then your solution must be called **ACW\_08346\_123456**. This solution will have to contain **at least** two projects:

- **ACW\_08346\_abcdef\_Client**: your client project software implementation
- **ACW\_08346\_abcdef\_Server**: your server project software implementation

You were told that **Windows Communication Foundation (WCF)** is the technology to use. You may use it or decide for a different technology. But note that even if you use a different technology, the result provided by the binary must be the same as if you used WCF. If you use WCF, then your solution will contain three projects, say if your Campus id is 123456, then these will be:

- **ACW\_08346\_123456\_Client**: your client project software implementation
- **ACW\_08346\_123456\_Server**: your server project software implementation
- **ACW\_08346\_123456\_ServiceLibrary**: your implementation of the WCF service library for the server

Your solutions **must be compileable** and it must be possible to build them to provide the following **executables in the following directories**. Make sure that **check this when building the entire solution in Visual Studio (as indicated above, substitute abcdef by your 6-digit Campus id)**:

*ACW\_08346\_abcdef\_Client*

*bin*

*Release*

***ACW\_08346\_abcdef\_Client.exe***

*ACW\_08346\_abcdef\_Server*

*bin*

*Release*

***ACW\_08346\_abcdef\_Server.exe***

Failure to conform to these criteria may result in the evaluation script **not finding your solution or its executables when automatically built, resulting in 0 marks being awarded by the evaluation system.**

## The Client

The client should read from standard input (the console) and should write to the standard output (the console). These will be as follows. **Follow these instructions carefully.**

The first line contains the number of next lines, e.g. 13 indicates that the input will consist of a line with number 13 and another 13 lines containing the instructions for the client. The next lines always consist of:

- ONE instruction;
- or **ONE instruction, ONE SPACE** and the data.

The instruction can be written in any case, i.e. both “Hello” and “HELLO” are acceptable. You may assume that input files will be correct, i.e. no handling of incorrect input is needed. The instructions that can occur are the following (substitute <argument> with an actual valid argument value):

- **HELLO <id>**: Will send a hello message to the server, containing <id>, which is a number (int) - a numerical identifier of the client to the server. The server then sends message **"Hello"** to the client.

Client output:

- Client displays the message it has obtained from the server (i.e. the **"Hello"**), terminated by **"\r\n"** (please note that **"\r\n"** is the Windows set of characters for a new line).
- **SORT <number> <value1> <value2> ... <valuen>**: Requests the server to sort the array, which contains <number> elements with **string** values <value1> <value2> ... <valuen>. Each pair of values should be separated by **exactly one space**. You may assume that the value <number> will not be higher than 100.

Client output:

- The client should then output the string **"Sorted values:\r\n"** and the list the sorted values obtained from the server, with each pair separated by **exactly one space**.
- **PUBKEY**: Requests the public key from the server; the server will respond by sending a two hexadecimal strings representing the public key parameters: **the exponent** and **the modulus**.

Client output:

- Both exponent and modulus should be output to the console window of the client in their hexadecimal forms, the first line represents the exponent and the second line represents the modulus.
- **ENC <message>**: Will encrypt the message <message> (this will be a string) using the **public key of the server** and send it to the server. The server then decrypts the message.

Client output:

- If no public key has been obtained (i.e. no PUBKEY instruction was specified prior to this call), the client will output: **"No public key.\r\n"** to the console.
- If the public key is known, then the client will output: **"Encrypted message sent.\r\n"** to the console.
- If a cryptographic exception is thrown, then the exception message should be displayed (**e.Message** for exception **e**), without an additional **"\r\n"**.
- Note that punctuation matters!
- **SHA1 <message>**: Will compute the SHA1 hash of message <message> (this will be a string).

Client output:

- The hash will be printed out in the **hexadecimal format** (i.e. as a sequence of 40 hexadecimal digits).
- **SHA256 <message>**: Will compute the SHA256 hash of message <message> (this will be a string).

Client output:

- The hash will be printed out in the **hexadecimal format** (i.e. as a sequence of 64 hexadecimal digits).

- **SIGN <message>**: Will request the server to **digitally sign** the message <message> (this will be a string) using the **private key of the server** and send it to the server. The server will then send the signed message back to the client. Your client will then verify this signature using the **public key of the server** and output the outcome of this process. Client output:
  - If no public key has been obtained (i.e. no PUBKEY instruction was specified prior to this call), the client will output: **"No public key.\r\n"** to the console.
  - If the public key is known, then the client will output: **"Signature successfully verified.\r\n"** to the console if the **signed obtained from the server was correct**. Otherwise, it will output: **"Data not signed.\r\n"** to the console.
  - If a cryptographic exception is thrown, then the exception message should be displayed (**e.Message** for exception **e**), without an additional **"\r\n"**.

## The Server

The server input will always consist of 8 lines (each terminated by **"\r\n"**). Each line can either be **empty** or **contain a hexadecimal string**. These hexadecimal strings represent the following components of the RSA public and private key (line by line, from the first to the last line):

DHex  
DPHex  
DQHex  
ExponentHex  
InverseQHex  
ModulusHex  
PHex  
QHex

All of these correspond to the respective fields of the **RSAParameters** struct:

```
public struct RSAParameters
{
    public byte[] D;
    public byte[] DP;
    public byte[] DQ;
    public byte[] Exponent;
    public byte[] InverseQ;
    public byte[] Modulus;
    public byte[] P;
    public byte[] Q;
}
```

Your server should load these hexadecimal strings from console input and convert them to the correct **byte[]** format. These will then be used for the encryption and digital signing operations.

The server should start by outputting this simple line into the console once started:

**"Server running..."**

The server should then wait in a separate thread for **less than 10 seconds**. During this period of time, server will listen for incoming connections of clients and perform the requested operations and send the results back to the clients in the form specified below. The server should automatically close after this period of time.

Once a request is obtained from a client, the server should respond in the following way, with the corresponding output provided to the console window of the server:

- **HELLO <id>**: String **"Hello"** should be sent back to the client. Server output:
  - The server should output **"Client No. <id> has contacted the server.\r\n"**, where <id> should be substituted by the actual id obtained from the client.
- **PUBKEY**: Two output parameters **representing the public key components** should be sent back to the client: **Exponent** and **Modulus**. It is up to you whether you send them as a byte array or the hexadecimal string, as long as your client can interpret and use these. Server output:
  - The server should output **"Sending the public key to the client.\r\n"**.

- **SORT <number> <value1> <value2> ... <valuen>**: It should also respond by sending the sorted values to the client.  
Server output:
  - The server should output **"Sorted values:\r\n"** and the list the sorted values in **ASCENDING** order (the default behaviour of `Array.Sort()` is expected), with each pair separated by **exactly one space**.
- **ENC <message>**: Encrypted data should be obtained from the client and decrypted. Server provides no response.  
Server output:
  - The server should output **"Decrypted message is: <message>.\r\n"**, where `<message>` is the result of the decryption of the encrypted data obtained from the data (using the private key).
  - If a cryptographic exception is thrown, then the exception message should be displayed (**e.Message** for exception **e**), without an additional **"\r\n"**.
- **SHA1 <message>**: The 40-byte hexadecimal string representing the SHA-1 hash of `<message>` should be sent to the client.  
Server output:
  - The server should output **"SHA-1 hash of <message> is <sha1hex>.\r\n"**, where `<message>` is the original message from the client and `<sha1hex>` is the 40-byte hexadecimal string representing the SHA-1 hash of `<message>`.
- **SHA256 <message>**: The 64-byte hexadecimal string representing the SHA-256 hash of `<message>` should be sent to the client.  
Server output:
  - The server should output **"SHA-256 hash of <message> is <sha256hex>.\r\n"**, where `<message>` is the original message from the client and `<sha256hex>` is the SHA-256 hash in the form of the hexadecimal string.
- **SIGN <message>**: The server should send a signed version of the message `<message>` to the client. The server's private key should be used to sign this message.  
Server output:
  - If `<message>` obtained from the client converted to UPPER CASE is **"CHEESECAKE"**, then the server should output **"No cheesecake allowed.\r\n"**.
  - Otherwise, the server should output **"Signing data: <message>.\r\n"**, where `<message>` is the message obtained from the client.
  - If a cryptographic exception is thrown, then the exception message should be displayed (**e.Message** for exception **e**), without an additional **"\r\n"**.

Three examples of typical server and client input and output are provided below. Feel free to use these to guide your development. Your distributed encryption service should provide exactly the same outputs for the specified inputs. You may use .NET API to perform all the cryptographic manipulations.

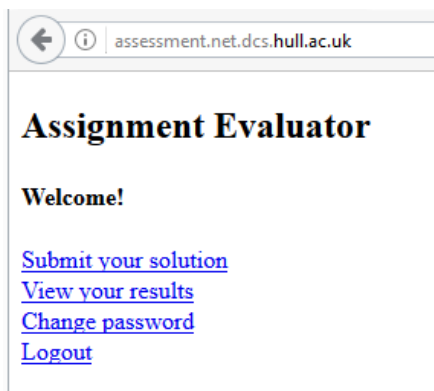
## Submission and Evaluation

Your solution must be provided in the structure specified above and must be submitted to **Canvas** as a ZIP file named **ACW\_08346\_abcdef.zip**, where **abcdef** has to be substituted by your actual **6-digit Campus id**. For example, if your Campus id is 123456, then your ZIP submission must be called **ACW\_08346\_123456.zip**.

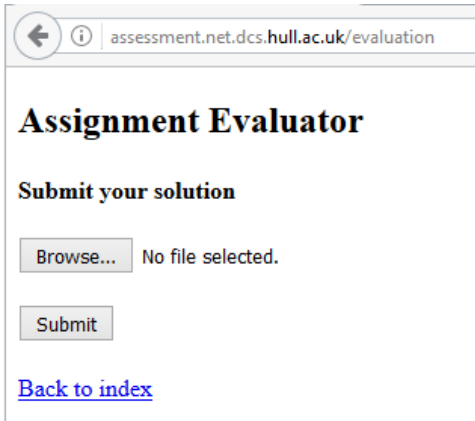
This solution must be tested and working on the machines in the Fenner lab, as well as at <http://assessment.net.dcs.hull.ac.uk>. **This Assessment Evaluator service is a transparent service provided to you to check your solution in advance against the tests.**

## Assignment Evaluator Instructions

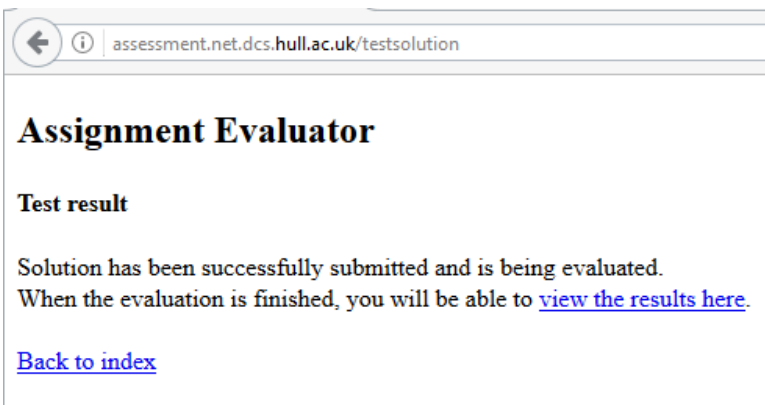
1. Use your 6-digit Campus id as a login and a **password that will be provided to you** (if you do not know the password yet, please contact David Chalupa <d.chalupa@hull.ac.uk>). **After signing in for the first time, change your password immediately.**



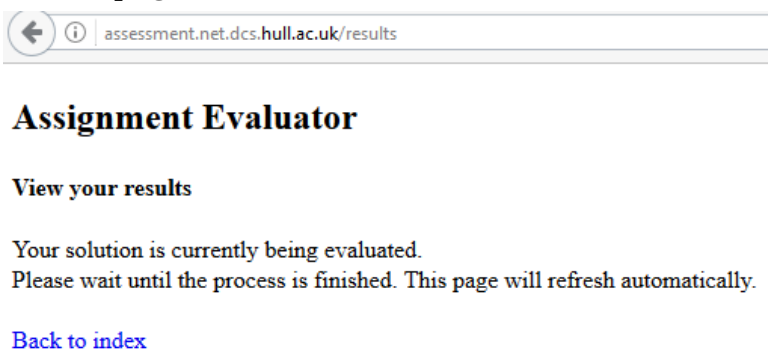
2. Click on *Submit your solution*. You will obtain a screen similar to the following. Use the file browser to select your ZIP file prepared according to the instructions above and click on Submit.



3. After the submission, you should get a screen similar to the following. Click on *view the results here*.



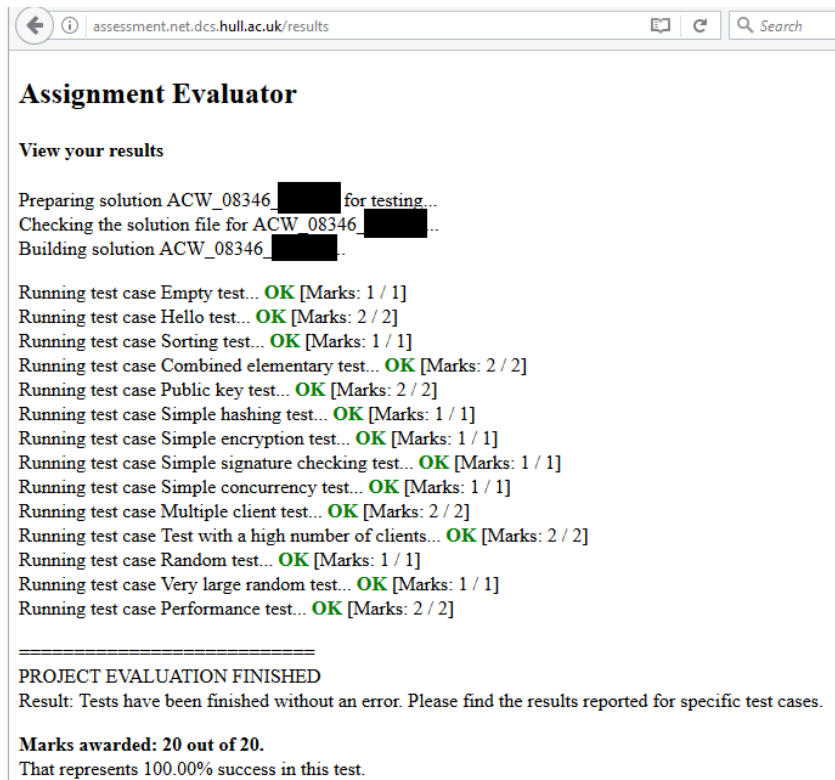
4. If your zipped solution had the correct structure, you will obtain a screen very similar to the following. This means that your solution is currently being evaluated. **This may take a long time so be patient. If you do not get any progress and still see this message after several hours, please contact David Chalupa <d.chalupa@hull.ac.uk>.**



5. When the process is finished, you should see a screen very similar to the following. **Please note that this is only a provisional mark, the final mark will be awarded by a standalone script used after the submission and by the module staff, on the basis of code and application inspection. Also, please be aware that only the last ZIP file that you submit gets archived by the Assignment Evaluator.**



If you obtained a different screen with an error message, please check if your Visual Studio solution has the correct structure and is compilable. Try resubmitting a new ZIP file with the correct solution and code structure. If you get an error that you think is caused by the evaluation system and not your solution, please contact module staff.



The screenshot shows a web browser window with the address bar displaying 'assessment.net.dcs.hull.ac.uk/results'. The page title is 'Assignment Evaluator'. Below the title, there is a section 'View your results'. The main content area shows the progress of the evaluation: 'Preparing solution ACW\_08346 [redacted] for testing...', 'Checking the solution file for ACW\_08346 [redacted] ..', and 'Building solution ACW\_08346 [redacted]..'. A list of test cases follows, each with a status 'OK' and marks: 'Running test case Empty test... OK [Marks: 1 / 1]', 'Running test case Hello test... OK [Marks: 2 / 2]', 'Running test case Sorting test... OK [Marks: 1 / 1]', 'Running test case Combined elementary test... OK [Marks: 2 / 2]', 'Running test case Public key test... OK [Marks: 2 / 2]', 'Running test case Simple hashing test... OK [Marks: 1 / 1]', 'Running test case Simple encryption test... OK [Marks: 1 / 1]', 'Running test case Simple signature checking test... OK [Marks: 1 / 1]', 'Running test case Simple concurrency test... OK [Marks: 1 / 1]', 'Running test case Multiple client test... OK [Marks: 2 / 2]', 'Running test case Test with a high number of clients... OK [Marks: 2 / 2]', 'Running test case Random test... OK [Marks: 1 / 1]', 'Running test case Very large random test... OK [Marks: 1 / 1]', and 'Running test case Performance test... OK [Marks: 2 / 2]'. A separator line is followed by the text 'PROJECT EVALUATION FINISHED'. Below this, it says 'Result: Tests have been finished without an error. Please find the results reported for specific test cases.' and 'Marks awarded: 20 out of 20.' with a note 'That represents 100.00% success in this test.'

## Evaluation Procedure

Your solution will be evaluated by:

- an automatic code testing routine (maximum 20 marks allocated; your solution can be preliminarily tested at <http://assessment.net.dcs.hull.ac.uk>);
- as well as inspection by the module staff (maximum 10 marks allocated).

Module staff reserve the right to adjust the automatic code testing routine or to perform an alternative evaluation of this part of the marking scheme. The marking will be carried out according to the marking scheme.

## Queries

If you have any questions about this exercise or would like a clarification, please send an email to both module staff: David Chalupa <d.chalupa@hull.ac.uk> and Nina Dethlefs <n.dethlefs@hull.ac.uk>.

## Sample Inputs and Outputs 1

Client Input

1

Hello 42

Server Input

[a file with 8 empty lines]

Client Output

Hello

Server Output

Server running...

Client No. 42 has contacted the server.

## Sample Inputs and Outputs 2

### Client Input

```
4
Hello 1
PubKey
PUBKEY
pubkey
```

### Server Input

[Warning: Note that 1<sup>st</sup> and 6<sup>th</sup> hexadecimal strings are both long strings that span two lines.]

```
0b537c55d90abc38f36c59d3b12b142c097e261999394a3ff6116941383cbf1da4e0634f7a95435dafbac669a005d337b4a7d97084ca93ddd6acd9ec735290996ae7bdac85c84d8b1f1e24c012bb8447
93f1acf5805d09ba421c4a8f58124f60754dcf615eda671c0d86233ff51656f141e05b89ee6302f30b8dc3bcd5bb2d41
31375be61215c75f69696db32ba6786723b7064b0ad69361ce8850fbbbed2c0620bf99352ea93cffe867231bc19d92a3ae3b8d86ba0e88b5133a6321ddc7ef41
265bcd4234a4d4d6887986728a1512a49bdeda70eaa50c59265ebd8e4d883aa6376727382f944734c75b3ce5ec1680bfe380da022f0f32e4f1b383740122497
010001
c64f02ba0562c196f9e528c55b4056cfdd06f240b1e21edee98d0c833eb08bfd8b2678f42402c32007f172abcd7b662fac423d0fcf1b56dc9b55a2a3247b4706
92644d841f0775d941fa5ba35aa3ccf65d7680135a719f47070fd8256fc0f54a7ead22191653f58be1d4d6ffa4b091af5d342f2cab1bed8ad82ca96722688e955e42b384261d1ccdf138e6d333496934
2db1a7005e3e0e26203eb72758d8161619ccd8ca71217234fac92d41e82d9b35a9a29c3dbd1d4c6e3496330c96aa69db
c92d8a37c1b9dd2dca44ad17b3625fbb78a241e9c28d78c3d9c6dcb2a2ef96d25b1a207c205c650465b57d67559ee471e41bf86fa294b291c73e94a84eb6e741
ba48ce81e9684e5f375fdf72df92abb5d64b33f455f03e947181bd8b6c70b101d2d44f7bc4ad3917684252f9e223125edae0697a0b70c76601cfeacceb54861b
```

### Client Output

```
Hello
010001
92644d841f0775d941fa5ba35aa3ccf65d7680135a719f47070fd8256fc0f54a7ead22191653f58be1d4d6ffa4b091af5d342f2cab1bed8ad82ca96722688e955e42b384261d1ccdf138e6d333496934
2db1a7005e3e0e26203eb72758d8161619ccd8ca71217234fac92d41e82d9b35a9a29c3dbd1d4c6e3496330c96aa69db
010001
92644d841f0775d941fa5ba35aa3ccf65d7680135a719f47070fd8256fc0f54a7ead22191653f58be1d4d6ffa4b091af5d342f2cab1bed8ad82ca96722688e955e42b384261d1ccdf138e6d333496934
2db1a7005e3e0e26203eb72758d8161619ccd8ca71217234fac92d41e82d9b35a9a29c3dbd1d4c6e3496330c96aa69db
010001
92644d841f0775d941fa5ba35aa3ccf65d7680135a719f47070fd8256fc0f54a7ead22191653f58be1d4d6ffa4b091af5d342f2cab1bed8ad82ca96722688e955e42b384261d1ccdf138e6d333496934
2db1a7005e3e0e26203eb72758d8161619ccd8ca71217234fac92d41e82d9b35a9a29c3dbd1d4c6e3496330c96aa69db
```

### Server Output

```
Server running...
Client No. 1 has contacted the server.
Sending the public key to the client.
Sending the public key to the client.
Sending the public key to the client.
```

## Sample Inputs and Outputs 3

### Client Input

```
6
Hello 42
SHA1 Hello world
Sha1
Sha256 Hah this is not fine
sha1 Oh my!
Hello 000
```

### Server Input

[a file with 8 empty lines]

### Client Output

```
Hello
7b502c3a1f48c8609ae212cdfb639dee39673f5e
da39a3ee5e6b4b0d3255bfef95601890afd80709
243cd820a2e0850175b58b82aca6fcc33d61942b2188da1368c714a3f5d0a365
6a18baeb0e51ecf93e06cc96984552d06d34115d
Hello
```

### Server Output

```
Server running...
Client No. 42 has contacted the server.
SHA-1 hash of Hello world is 7b502c3a1f48c8609ae212cdfb639dee39673f5e.
SHA-1 hash of  is da39a3ee5e6b4b0d3255bfef95601890afd80709.
SHA-256 hash of Hah this is not fine is 243cd820a2e0850175b58b82aca6fcc33d61942b2188da1368c714a3f5d0a365.
SHA-1 hash of Oh my! is 6a18baeb0e51ecf93e06cc96984552d06d34115d.
Client No. 0 has contacted the server.
```