Chapter 6

Socket for Clients

Content

1. Using Sockets
    i. Investigating Protocols with Telnet
    ii. Reading from Servers with Sockets
    iii. Writing to Servers with Sockets
2. Constructing and Connecting Sockets
    i. Basic Constructors
    ii. Picking a Local Interface to Connect From
    iii. Constructing Without Connecting
    iv. Socket Addresses
    v. Proxy Servers
3. Getting Information About a Socket
    i. Closed or Connected?
    ii. toString()
4. Setting Socket Options
    i. TCP_NODELAY
    ii. SO_LINGER

iii. SO_TIMEOUT

iv. SO_RCVBUF, SO_SNDBUF

v. SO_KEEPALIVE

vi. SO_OOBINLINE

vii. SO_REUSEADDER

viii. IP_TOS

5. Socket in GUI Applications: Whois, and A Network Client Library

Using Sockets

A socket is a connection between two hosts. It can perform seven basic operations:

- Connect to a remote machine
- Send data
- Receive data
- Close a connection
- Bind to a port (only server side)
- Listen for incoming data (only server side)
- Accept connections from remote machines on the bound port (only server side)

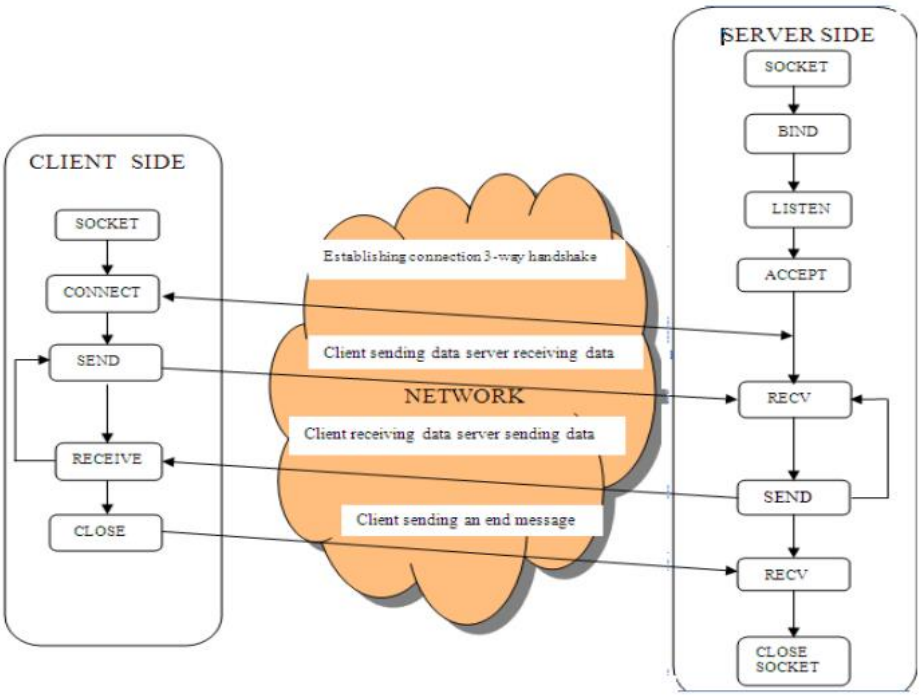

Fig: TCP Client-Server

- Normally there are 7 different operations that are performed in a client-server communication
- The above 4 operations are performed on both client and server side, whereas the last 3 operations are performed on server side only
- **Socket**, and **ServerSocket** are two classes in Java responsible for implementing the client side and server side sockets respectively

Java programs normally use **client Sockets** in the following fashion:

1. The program creates a new socket with a constructor.
2. The socket attempts to connect to the remote host.

Once the connection is established,

- the local and remote hosts get input and output streams from the socket and use those **streams** to send data to each other
- This connection is **full-duplex**
- When the transmission of data is complete, one or both sides **close** the connection.

Investigating Protocols with Telnet

- To get a feel for how a protocol operates, you can use Telnet to connect to a server,
- type different commands to it, and watch its responses.
- By default, Telnet attempts to connect to port 23.
- To connect to servers on different ports, specify the port you want to connect to like this:

  $ telnet localhost 25
- This requests a connection to port 25, the SMTP port, on the local machine

Reading from Servers with Sockets

- To connect to the daytime server at the National Institute for Standards and Technology (NIST) and ask it for the current time.

telnet time.nist.gov 13

Trying 129.6.15.28...

Connected to time.nist.gov.

Escape character is '^]'.

56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *

Connection closed by foreign host

the format is defined as JJJJJ YY-MM-DD HH:MM:SS TT L H msADV UTC(NIST) OTM where:

- JJJJJ is the "Modified Julian Date" (i.e., it is the number of whole days since midnight on November 17, 1858).
- YY-MM-DD is the last two digits of the year, the month, and the current day of month.
- HH:MM:SS is the time in hours, minutes, and seconds in Coordinated Universal Time (UTC, essentially Greenwich Mean Time).
- TT indicates whether the United States is currently observing on **Standard Time or Daylight Savings** Time: 00 means standard time; 50 means daylight savings time. Other values count down the number of days until the switchover.
- L is a one-digit code that indicates whether a **leap** second will be added or subtracted at midnight on the last day of the current month: 0 for no leap second, 1 to add a leap second, and 2 to subtract a leap second.
- H represents the **health of the server**: 0 means healthy, 1 means up to 5 seconds off, 2 means more than 5

seconds off, 3 means an unknown amount of inaccuracy, and 4 is maintenance mode.

- msADV is a number of **milliseconds** that NIST adds to the time it sends to roughly **compensate for network delays**. In the preceding code, you can see that it added 888.8 milliseconds to this result, because that's how long it estimates it's going to take for the response to return.
- UTC(NIST) is a constant, and the OTM is almost a constant (an asterisk

unless something really weird has happened).

To read the same using socket programming,

- First, open a socket to time.nist.gov on port 13:

  Socket socket = new Socket("time.nist.gov", 13);

This creates the object and makes the connection across the network.

- Now, call **getInputStream**() to return an **InputStream** you can use to read bytes from the socket.

  InputStream in = socket.getInputStream();

```
StringBuilder time = new StringBuilder();
InputStreamReader reader = new
InputStreamReader(in, "ASCII");
        for (int c = reader.read(); c != -1; c =
reader.read()) {
        time.append((char) c);
    }
    System.out.println(time);
```

Example: A daytime protocol client

```
import java.net.*;
import java.io.*;
public class DaytimeClient {
public static void main(String[] args) {
String hostname = "time.nist.gov";
Socket socket = null;
try {
socket = new Socket(hostname, 13);
socket.setSoTimeout(15000);
InputStream in = socket.getInputStream();
StringBuilder time = new StringBuilder();
InputStreamReader reader = new InputStreamReader(in,
"ASCII");
```

```
for (int c = reader.read(); c != -1; c = reader.read()) {
time.append((char) c);
}
System.out.println(time);
} catch (IOException ex) {
System.err.println(ex);
} finally {
if (socket != null) {
try {
socket.close();
} catch (IOException ex) {
// ignore
}
}
}
}
}
```

Typical output is much the same as if you connected with Telnet:

```
$ java DaytimeClient
56375 13-03-24 15:05:42 50 0 0 843.6 UTC(NIST) *
```

Writing to Servers with Sockets

In the most common pattern, the client sends a request. Then the server responds. The client may send another request, and the server responds again. This continues until one side or the other is done, and closes the connection.

- Let say there is a server listing at “fakedict.org”, it will give defination of any english word.
- To do so, First, open a socket to a dict server—dict.org__ is a good one—on port 2628:
- *Socket socket = new Socket("fakedict.org", 2628);*
- you’ll want to set a timeout in case the server hangs while you’re connected to it:
- *socket.setSoTimeout(15000);*
- the client speaks first, so ask for the output stream using getOutputStream():

  OutputStream out = socket.getOutputStream();

  it’s convenient to wrap this in a Writer

  Writer writer = new OutputStreamWriter(out, "UTF-8");
- Now write the command over the socket:
- *writer.write("DEFINE eng-lat gold\r\n");*
- Finally, flush the output so you’ll be sure the command is sent over the network:
- *writer.flush();*

- The server should now respond with a definition. You can read that using the socket's input stream.

Half-closed sockets

- The **close**() method shuts down both input and output from the socket.
- On occasion, you may want to shut down only half of the connection, either input or output.
- The shutdownInput() and shutdownOutput() methods close only half the connection:

  public void shutdownInput() throws IOException

  public void shutdownOutput() throws IOException

- **Neither actually closes the socket**. Instead, they adjust the stream connected to the socket so that it thinks it's at the end of the stream
- Further reads from the input stream after shutting down input **return –1.**
- Further writes to the socket after shutting down output throw an **IOException**.
- Note that even though you shut down half or even both halves of a connection, you still need to close the socket when you're through with it.

- The shutdown methods only affect the socket's streams.
- They don't release the resources associated with the socket, such as the port it occupies.

Constructing and Connecting Sockets

The java.net.Socket class is Java's fundamental class for performing client-side TCP operations.

1. Basic Constructors

- Each Socket constructor specifies the host and the port to connect to. Hosts may be specified as an InetAddress or a String. Remote ports are specified as int values from 1 to 65535

1. public Socket(String host, int port) throws UnknownHostException, IOException

- This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
- If the connection can't be opened for some reason, the constructor throws an **IOException** or an **UnknownHostException**.

Q. Find out which of the first 1024 ports seem to be hosting TCP servers on a specified host (say, localhost)

```
import java.net.*;
import java.io.*;
public class LowPortScanner {
public static void main(String[] args) {
String host = args.length > 0 ? args[0] : "localhost";
for (int i = 1; i < 1024; i++) {
try {
Socket s = new Socket(host, i);
System.out.println("There is a server on port " + i + " of "
+ host);
s.close();
} catch (UnknownHostException ex) {
System.err.println(ex);
break;
} catch (IOException ex) {
// must not be a server on this port
}
}
}
}
```

2. public Socket(InetAddress host, int port) throws IOException

- ➢ This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.

3. public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.

- ➢ Connects to the specified host and port, creating a socket on the local host at the specified address and port.

4. public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.

- ➢ This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.

5. public Socket()

- ➢ Creates an unconnected socket. Use the connect() method to connect this socket to a server.

6. public Socket(Proxy proxy)

2. Picking a Local Interface to Connect From

- Two constructors specify both the host and port to connect to and the interface and port to connect from:

  public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException

  public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException

- This socket connects to the host and port specified in the first two arguments.
- It **connects from the local network interface** and port specified by the last two arguments.
- The network interface **may be either physical** (e.g., an Ethernet card) or **virtual** (a multi-homed host with more than one IP address).
- If 0 is passed for the localPort argument,Java chooses a **random available port** between 1024 and 65535.
- For example:

  try {

  String myLocalIP = "192.168.72.68";

  InetAddress localInterface = InetAddress.getByName(myLocalIP);

Socket socket = new Socket("mail", 25, localInterface, 0);

// work with the sockets...

} catch (IOException ex) {

System.err.println(ex);

}

Constructing Without Connecting

- All the constructors we've talked about so far both create the socket object and open a network connection to a remote host. Sometimes you want to split those operations.
- If you give no arguments to the Socket constructor, it has nowhere to connect to:

  public Socket()
- You can connect later by passing a **SocketAddress** to one of the connect() methods.

  For example:

  try {

  Socket socket = new Socket();

  // fill in socket options

  SocketAddress address = new

  ***InetSocketAddress**("time.nist.gov", 13);*

```
        socket.connect(address, 5000);
        // work with the sockets...
    } catch (IOException ex) {
        System.err.println(ex);
    }
```

- You can pass an int as the second argument to specify the number of milliseconds to wait before the connection times out:

  public void connect(SocketAddress endpoint, int timeout) throws IOException

- **The default, 0, means wait forever**.

Socket Addresses: Reusing the Sockets

- The **SocketAddress** class represents a connection endpoint.
- The **SocketAddress** class can be used for both TCP and non-TCP sockets.
- In practice, only TCP/IP sockets are currently supported and the socket addresses you actually use are all instances of InetSocketAddress
- The primary purpose of the **SocketAddress** class is to provide a convenient store for transient socket

connection information such as the IP address and port that can be reused to create new sockets, even after the original socket is disconnected and garbage collected.

- the Socket class offers two methods that return SocketAddress objects

i. getRemoteSocketAddress(): returns the address of the system being connected

ii. getLocalSocketAddress(): returns the address from which the connection is made

public SocketAddress getRemoteSocketAddress()

public SocketAddress getLocalSocketAddress()

- Both of these methods return null if the socket is not yet connected.
- For example, first you might connect to Yahoo! then store its address:

```
Socket socket = new Socket("www.yahoo.com", 80);
SocketAddress yahoo = socket.getRemoteSocketAddress();
socket.close();
```

- Later, you could reconnect to Yahoo! using this address:

```
Socket socket2 = new Socket();
socket2.connect(yahoo)
```

Proxy Servers

- The another type of constructor creates an unconnected socket that connects through a specified proxy server

```
public Socket(Proxy proxy)
```

- Normally, the proxy server a socket uses is controlled by the **socksProxyHost** and **socksProxyPort** system properties, and these properties apply to all sockets in the system
- For example, this code fragment uses the SOCKS proxy server at myproxy.example.com to connect to the host login.ibi-blio.org:

```
SocketAddress proxyAddress = new
InetSocketAddress("myproxy.example.com", 1080);
```

```
Proxy proxy = new Proxy(Proxy.Type.SOCKS,
proxyAddress)
Socket s = new Socket(proxy);
SocketAddress remote = new
InetSocketAddress("login.ibiblio.org", 25);
s.connect(remote);
```

Getting Information About a Socket

Socket objects have several properties that are accessible through getter methods:

- Remote address
- Remote port
- Local address
- Local port

Here are the getter methods for accessing these properties:

- public InetAddress getInetAddress()
- public int getPort()
- public InetAddress getLocalAddress()
- public int getLocalPort()

➢ The **getInetAddress**() and **getPort**() methods tell you the remote host and port the Socket is connected to; or,

if the connection is now closed, which host and port the Socket was connected to when it was connected.

- The **getLocalAddress()** and **getLocalPort()** methods tell you the network interface and port the Socket is connected from
- There are no setter methods. These properties are set as soon as the socket connects and are fixed from there on.

Example:

```
import java.net.*;
import java.io.*;

public class SocketInfo {
    public static void main(String[] args) {
        for (String host : args) {
            try {
                Socket theSocket = new Socket(host, 80);
                System.out.println(
                "Connected to " + theSocket.getInetAddress()
                + " on port " + theSocket.getPort() + " from port "
```

```
                    + theSocket.getLocalPort() + " of "
                    + theSocket.getLocalAddress());
                } catch (UnknownHostException ex) {
                    System.err.println("I can't find " +
host);
                } catch (SocketException ex) {
                    System.err.println("Could not connect
to " + host);
                } catch (IOException ex) {
                    System.err.println(ex);
                }
            }
        }
}
```

Closed or Connected

- The isClosed() method returns true if the socket is closed, false if it isn't
- For example:

```
if (socket.isClosed()) {
// do something...
} else {
```

// do something else...

}

- However, this is not a perfect test. If the socket has never been connected in the first place, isClosed() returns false, even though the socket isn't exactly open.
- The Socket class also has an isConnected() method. The name is a little misleading.
- It does not tell you if the socket is currently connected to a remote host
- Instead, it tells you whether the socket has ever been connected to a remote host
- If the socket was able to connect to the remote host at all, this method returns true, even after that socket has been closed
- Therefore, to know **if the socket is currently connected**, you have to call both isConnected() and isClosed() methods, in response to which isConnected() returns true and isClosed() returns false.

```
boolean connected = socket.isConnected() && !
socket.isClosed()
```

- there is another method **isBound()** that tells you whether the socket successfully bound to the outgoing port on the local system

- Note that, isConnected() refers to the remote end of the socket, isBound() refers to the local end.

toString()

- The toString() method produces a string that looks like this

  Socket[addr=www.oreilly.com/198.112.208.11,port=80,localport=50055]

- This is useful primarily for debugging

Setting Socket Options

1. **TCP_NODELAY**
   - The TCP_NODELAY socket option is used to disable the Nagle algorithm, which is a congestion-control algorithm used by TCP to reduce network traffic.
   - When Nagle's algorithm is enabled, TCP tries to reduce the number of small packets sent over the network by buffering small amounts of data until there is enough to fill a packet.
   - This can improve performance in some cases, but can also introduce delays in the delivery of data, especially for real-time applications or interactive services.

- This can be useful for applications that require low latency and real-time responsiveness, such as online gaming, video conferencing, or other interactive services.
- In java, TCP_NODELAY option can be set or get on a Socket or ServerSocket object using the
  - public void setTcpNoDelay(boolean on) throws SocketException
  - public boolean getTcpNoDelay() throws SocketException
- setTcpNoDelay(true) turns off buffering for the socket.
- setTcpNoDelay(false) turns it back on.
- getTcpNoDelay() returns true if buffering is off and false if buffering is on.
- Example

  Socket socket = new Socket("example.com", 80);

  socket.setTcpNoDelay(true);

2. SO_LINGER

- The SO_LINGER option specifies what to do with datagrams that have not yet been sent when a socket is closed.

- By default, the close() method returns immediately; but the system still tries to send any remaining data.
- If the linger time is set to zero, any unsent packets are thrown away when the socket is closed.
- If SO_LINGER is turned on and the linger time is any positive value, the close() method blocks while waiting the specified number of seconds for the data to be sent and the acknowledgments to be received.
- When that number of seconds has passed, the socket is closed and any remaining data is not sent.
- In java, SO_LINGER option can be set or get on a Socket or ServerSocket object using the
  - public void setSoLinger(boolean on, int seconds) throws SocketException
  - public int getSoLinger() throws SocketException

- These two methods each throw a **SocketException** if the underlying socket implementation does not support the SO_LINGER option.
- Example:

  Socket socket = new Socket("example.com", 80);

  socket.setSoLinger(true, 10);

3. Timeout

- The SO_TIMEOUT socket option, also known as the socket timeout, is used to control the amount of time a socket will block on certain socket operations before throwing a timeout exception.
- The socket timeout is a value in milliseconds that specifies the maximum amount of time that the socket will wait for an operation to complete.
- For example, when you try to read data from a socket, the read() call blocks as long as necessary to get enough bytes.
- By setting SO_TIMEOUT, you ensure that the call will not block for more than a fixed number of milliseconds. When the timeout expires, an InterruptedIOException is thrown,
- In java, SO_TIMEOUT option can be set or get on a Socket or ServerSocket object using the
  - public void setSoTimeout(int milliseconds) throws SocketException
  - public int getSoTimeout() throws SocketException
- Example:

  Socket socket = new Socket("example.com", 80);

  socket.setSoTimeout(5000);

4. SO_RCVBUF and SO_SNDBUF

- SO_RCVBUF and SO_SNDBUF are socket options used to control the size of the socket receive buffer and socket send buffer, respectively.
- The socket receive buffer is used to hold incoming data from the network until it can be processed by the receiving application.
- The socket send buffer is used to hold outgoing data from the application until it can be sent over the network.
- In Java, these options can be set on a Socket or ServerSocket object using the setReceiveBufferSize() and setSendBufferSize() methods
  - ➢ public void setReceiveBufferSize(int size) throws SocketException, IllegalArgumentException
  - ➢ public void setSendBufferSize(int size) throws SocketException, IllegalArgumentException
- Also, these options can be get on a Socket or ServerSocket object using the setReceiveBufferSize() and setSendBufferSize() methods
  - ➢ public int getReceiveBufferSize() throws SocketException

➢ public int getSendBufferSize() throws SocketException

- Example,

  Socket socket = new Socket("example.com", 80);

  socket.setReceiveBufferSize(8192);

  socket.setSendBufferSize(8192);

5. SO_KEEPALIVE

- The SO_KEEPALIVE socket option is used to enable or disable the sending of TCP keepalive messages on a socket connection.
- TCP keepalive messages are packets sent periodically by a connection to check if the other end of the connection is still responsive.
- When SO_KEEPALIVE is enabled on a socket, the operating system will periodically send keepalive messages on the connection, and if no response is received, it will close the connection.
- This helps to detect when a connection has been dropped or when the remote end has become unresponsive, even if no data is being sent or received.
- The default for SO_KEEPALIVE is false

- In Java, the SO_KEEPALIVE option can be set on a Socket object using the setKeepAlive() method.
  - public void setKeepAlive(boolean on) throws SocketException
  - public boolean getKeepAlive() throws SocketException
- Example,

  Socket socket = new Socket("example.com", 80);

  socket.setKeepAlive(true);

6. OOBINLINE

- TCP includes a feature that sends a single byte of "urgent" **data out of band.**
- This data is sent immediately.
- Furthermore, the receiver is notified when the urgent data is received and may elect to process the urgent data before it processes any other data that has already been received.
- Java supports both sending and receiving such urgent data.
- The sending method is named, **sendUrgentData()**

  public void sendUrgentData(int data) throws IOException

- By default, Java ignores urgent data received from a socket. However, if you want to receive urgent data inline with regular data, you need to set the OOBINLINE option to true using these methods

  public void setOOBInline(boolean on) throws SocketException

  public boolean getOOBInline() throws SocketException

- Example:

  Socket socket = new Socket("example.com", 80);

  socket.setOOBInline(true);

7. SO_REUSEADDR

- When a socket is closed, it may not immediately release the local port, especially if a connection was open when the socket was closed.
- It can sometimes wait for a small amount of time to make sure it receives any lingering packets that were addressed to the port that were still crossing the network when the socket was closed
- If the SO_REUSEADDR is turned on (it's turned off by default), another
- socket is allowed to bind to the port even while data may be outstanding for the previous socket.

- In Java this option is controlled by these two methods:
  - public void setReuseAddress(boolean on) throws SocketException
  - public boolean getReuseAddress() throws SocketException
- Example,

  ServerSocket serverSocket = new ServerSocket(8080);

  serverSocket.setReuseAddress(true);

8. IP_TOS Class of Service

- Different types of Internet service have different performance needs.
- For instance, video chat needs relatively high bandwidth and low latency for good performance, whereas email can be passed over low-bandwidth connections and even held up for several hours without major harm
- The class of service is stored in an eight-bit field called IP_TOS in the IP header. Java lets you inspect and set the value a socket places in this field using these two methods:

  public int getTrafficClass() throws SocketException

public void setTrafficClass(int trafficClass) throws SocketException

- **The traffic class is given as an int between 0 and 255.**
- As an alternative way to express preferences, the setPerformancePreferences() method assigns relative preferences to connection time, latency, and bandwidth:

  public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)

- **Among three arguments, whichever has the highest value, is most important characteristic.**
- For instance, if connectionTime is 2, latency is 1, and bandwidth is 3, then maximum bandwidth is the most important characteristic, minimum latency is the least important, and connection time is in the middle.
- If connectionTime is 2, latency is 2, and band width is 3, then maximum bandwidth is the most important characteristic, while minimum latency and connection time are equally important

Whois Protocol

- Whois is a simple directory service protocol defined in RFC 954; it was originally designed to keep track of administrators responsible for Internet hosts and domains.
- The basic structure of the whois protocol is
  1. The client opens a TCP socket to port 43 on the server.
  2. The client sends a search string terminated by a carriage return/linefeed pair (\r\n). The search string can be a name, a list of names, or a special command, as discussed shortly. You can also search for domain names, like www.oreilly.com or net-scape.com, which give you information about a network.
  3. The server sends an unspecified amount of human-readable information in response to the command and closes the connection.
  4. The client displays this information to the user
- Example:

$ whois merojob.com

Domain Name: MEROJOB.COM

Registry Domain ID: 567869007_DOMAIN_COM-VRSN

Registrar WHOIS Server: whois.tucows.com
Registrar URL: http://www.tucows.com
Updated Date: 2022-08-08T09:27:23Z
Creation Date: 2006-08-24T14:29:19Z
Registry Expiry Date: 2024-08-24T14:29:19Z
Registrar: Tucows Domains Inc.
Registrar IANA ID: 69
Registrar Abuse Contact Email: domainabuse@tucows.com
Registrar Abuse Contact Phone: +1.4165350123
Domain Status: ok https://icann.org/epp#ok
Name Server: CASH.NS.CLOUDFLARE.COM
Name Server: FAY.NS.CLOUDFLARE.COM
DNSSEC: unsigned

There are different Prefixes we can use with whois:

Prefix	Meaning
Domain	Find only domain records.
Gateway	Find only gateway records.
Group	Find only group records.
Host	Find only host records.
Network	Find only network records.
Organization	Find only organization records.
Person	Find only person records.
ASN	Find only autonomous system number records.
Handle or !	Search only for matching handles.
Mailbox or @	Search only for matching email addresses.
Name or :	Search only for matching names.
Expand or *	Search only for group records and show all individuals in that group.
Full or =	Show complete record for each match.
Partial or suffix	Match records that start with the given string.
Summary or $	Show just the summary, even if there's only one match.
SUBdisplay or %	Show the users of the specified host, the hosts on the specified network, etc.