Protocols notes.

General.
HTTP, HTTPS, IIOP and JRMP all require a reliable, connection-based transport.
Currently all use TCP.

Common service ports.

Ping	- 7
HTTP/HTTPS	- 80 / 443
FTP	- 20 (data) and 21 (control)
Telnet	- 23
SMTP	- 25
POP3	- 110
JRMP	- naming service uses 1099; object servers allocated ports dynamically
IIOP	- object servers allocated ports dynamically
LDAP/LDAPS	- 389 / 636
DNS	- 53
DHCP	- 67
NNTP	- 119
NetBios	- 135, 137-139, 445
IMAP	- 143
SNMP	- 161 and 162 (trap)

Scenarios.
HTTP/HTTPS – internet sites
RMI-IIOP – intranet environment; interoperability requirements (EJB-EJB, EJB-CORBA, etc.)
RMI-JRMP – intranet with an all Java environment

Firewalls
Firewalls provide protection by :

- packet filtering
  IP blocking – source or destination IP
  port blocking – e.g. only allow well-known ports such as 80, 443, 25, etc.

- protocol filtering – e.g. no FTP

HTTP/HTTPS aren't generally affected by firewalls as ports 80 and 443 are normally allowed through. However, HTTP clients cannot generally accept asynchronous updates (push model) as more than likely the client will be prevented from running a socket server.

Most of the time, IIOP and JRMP are affected by firewalls as they use dynamic port allocation for object servers and use less well-known ports.
Although IIOP and JRMP can be tunnelled through HTTP this isn't ideal – the main benefit of being connection-based protocols is then lost.

HTTP (HyperText Transfer Protocol).
Stateless, connection-less request/response mechanism, default port is ***80***.
Regarded as connection-less :

- HTTP1.0 uses *tear-down*
  Client opens/closes socket connection to the web server every time

- HTTP1.1 uses *keep-alive* (a persistent connection).
  Client holds socket connection open over multiple requests.
  Still regarded as connection-less as it's acceptable for the client or server to drop the connection at any time (e.g. Apache uses a 15 sec timeout on keep-alive sockets), client will re-establish the connection.
  Client/server can use pipelining – i.e. sends multiple requests/responses down the socket at once.

A HTTP frame is composed of the following (each separated by CR+LF) :

- detail line
  ```
  GET /index.html HTTP/1.0    (request)
  HTTP/1.0 302 Found          (response)
  ```

- headers (name/value pairs)
  ```
  Date: Wed, 13 Mar 2002 21:30:48 GMT
  ```

- message body (optional)
  ```
  <HTML>Hello</HTML>
  ```

Valid request types : `GET, POST, HEAD, PUT, OPTIONS, DELETE, TRACE, CONNECT`

Pros	***Cons***
• simple request/response mechanism • allowed through firewalls (most of the time) • widely supported / deployed • extensible - supports tunnelling of arbitrary data; custom request types/response content	• stateless - hence non-transactional • insecure - hence HTTPS • inefficient - e.g. MIME encoding can make files bigger • doesn't support "push" model

HTTPS.
SSL (Secure Socket Layer) is an application level protocol layered over TCP.
HTTP layered over SSL is HTTPS, default port is ***443***.

Regarded as *connection-based / stateful* as an *SSL session* is maintained over multiple requests/responses. Each session may include multiple secure connections. In addition, each party (client/server) may hold multiple SSL sessions.

When a client and server first begin to communicate, they create an *SSL session* via an *SSL handshake* :

- 1. the client and server exchange capabilities (e.g. SSL version, ciphers available, etc.). The server includes it's certificate in the response
- 2. client verifies the CA's digital signature in the server certificate using the CA's public key
- 3. if OK, the client generates a *pre-master secret*, encrypts it using the server's public key and sends it to the server

  [if using client certificates, the client response includes the pre-master secret and the client certificate. The server verifies the CA's digital signature in the client certificate using the CA's public key]

- 4. the server decrypts the pre-master secret using it's private key
- 5. using the pre-master secret, the client and server use an agreed algorithm to independently generate the *master secret*
- 6. the master secret is then used to generate a set of *session keys* (symmetric keys for speed)
- 7. the client and server exchange messages confirming that the encrypted session has started. Subsequent SSL records are compressed and encrypted / signed using the session keys

Pros	***Cons***
• secure – only "in-the-clear" at client / server • allowed through firewalls (most of the time) • widely supported / deployed	• computationally expensive – excluding dedicated hardware, HTTPS can only process 10% of the traffic HTTP can • admin overhead – renewing certs, etc.

IIOP (Internet Inter-ORB Protocol).
The GIOP (General Inter-ORB Protocol) specifies a set of message formats and common data representations for communication and is intended for use on any suitable *connection-based* transport protocol.
IIOP is GIOP layered over TCP, *no default port* – the GIOP/IIOP implementation will dynamically assign ports when an object server instance binds to a name.

GIOP/IIOP has the following features :

- request/response based
- supports propagation of security/transactional context
- primarily used for invoking methods on remote objects in a *type-safe* manner and returning the result of the invocation via request/response *marshalling* – i.e. serializing, transmitting to/from client/server
- an open standard from the OMG
- mostly used for intranets / extranets – i.e. clients behind corporate firewalls or VPN connections

Clients communicate with servers via an *IOR* (Interoperable Object Reference).
The IOR contains a set of profiles (details of host, port, object instance, etc.) so that a client knows where to connect / send messages back and forth.

Java RMI can now operate with IIOP as the underlying protocol.
This enables non-Java clients to invoke methods on Java object servers (with some restrictions).

Pros	***Cons***
• Interoperability – Java clients can call C++ servers; Cobol client can call Java server, etc. • Legacy integration • Designed for generic remote object invocation - type safe, extensible • Inbuilt support for security and transactions	• Firewalls – servers bound to arbitrary ports so firewall can't be configured with an IIOP port; can be supported with an IIOP proxy but "push" model is still problematic • Performance – all method invocations are remote (possibly over the internet) • Pass-by-reference – only supports remote references, can't use pass-by-value (until CORBA 2.3)

JRMP (Java Remote Method Protocol).
JRMP is the default protocol for RMI – *stateful / connection-based.*
To enable interoperability, RMI has been extended to support RMI over IIOP (with some restrictions).
As of EJB1.1, all application servers must support RMI-IIOP.

The default port for the RMI registry / JNDI name server is 1099, object servers will be dynamically assigned ports by the RMI runtime.

RMI has the following features :

- supports dynamic download of code stubs via the RMI registry
- remote objects (implement java.rmi.Remote) are passed by reference (passes a copy of the stub)
- non-remote objects by passed by value (object is copied via serialization)
- distributed garbage collection
- the RMI client provides inbuilt support for HTTP tunnelling – e.g. if cannot connect on to object server, wrap request up as a HTTP POST and send to port 80 of the host.
- RMI servers provide support for HTTP POST request RMI over HTTP but require a proxy – e.g. an RMI servlet

Pros	***Cons***
• Richer feature set than IIOP – pass by reference or by value; distributed GC; stub download • Single multiplexed connection	• Interoperability - only works in a Java environment • Lacks IIOP's support for transaction/security context propagation • Firewalls – servers bound to arbitrary ports; HTTP tunnelling limited • Performance – all method invocations are remote (possibly over the internet)