The JPIK Protocol (JPeg2000 Interactive, Kakadu)

David Taubman, UNSW

November 30, 2001

1 Overview

"JPIK" is the name given to the protocol used to communicate between the Kakadu server and a conforming client. The current Kakadu software package offers the "kdu-show" utility as a suitable conforming client. Where the protocol is to be identified in a URL, the URL shall commence with the prefix, *jpik:*, as in *"jpik://my.machine.unsw.edu.au/nice-image.jp2"*.

The purpose of the JPIK protocol is to demonstrate one of the key capabilities of the EBCOT paradigm underlying JPEG2000. Specifically, the compression paradigm offers a degree of spatial random access to the compressed data source, in addition to access by resolution, by quality and by image components of interest. These four dimensions of scalability make JPEG2000 particularly suitable for interactive client-server applications.

In client-server applications, a client dynamically identifies its spatial region, resolution and image components of interest to the server, and the server delivers those portions of the compressed image representation which are relevant to the client, in an order which corresponds to progressively improving image quality over the relevant region and resolution of interest. Applications of this form were envisaged, and indeed advertised, as key motivators for the adoption of the EBCOT compression paradigm into the JPEG2000 standard in 1998. JPIK provides a convincing public demonstration of these capabilities.

1.1 Tiles vs. Precincts and Code-Blocks

In JPEG2000, the term *"tile,"* refers to a rectangular region of the image which is compressed independently, as though it were a separate image. To emphasize this point, we note that that any tile may be extracted from a JPEG2000 compressed code-stream and rewritten as a new code-stream in its own right. Evidently, tiles provide a means for spatial random access into an image. The *"Flashpix"* format uses a tiling strategy in combination with JPEG to achieve spatial random access. An important distinction between Flashpix and JPEG2000 is that each independently compressed tile has its own resolution

hierarchy and, at any given resolution, its own progressively refineable quality hierarchy.

A more important distinction between Flashpix and JPEG2000 is that spatial random access may be obtained even without resorting to tiling. We use the term *"untiled"* to refer to an image which has only one tile. Such an image is compressed as a whole, thereby avoiding the tile-boundary artefacts commonly encountered with tiling. The fact that untiled JPEG2000 images offer spatial random access is a direct consequence of the EBCOT paradigm (Embedded Block Coding with Optimized Truncation), whereby the untiled image subbands[1] are partitioned into smaller blocks of transform coefficients, known as *"code-blocks,"* each of which is represented with an independent, finely embedded code.

At a slightly higher level, JPEG2000 arranges the code-blocks into so-called *"precincts,"* and offers a quality-embedded representation for each precinct through so-called *"packets."* These packets are to be distinguished from network packets. A precinct represents a spatial region within a given *"resolution level"* of a given image component. Each resolution level builds upon the previous resolution level to offer the additional details required for increased reconstructed image resolution.

Code-blocks and precincts are each identifiable elements within a JPEG2000 code-stream, which correspond to localized spatial regions of the image, each belonging to one image component at one resolution (or one spatial frequency subband), and each having a quality embedding which allows progressive refinement of the reconstructed image quality. By accessing compressed data from particular code-blocks or particular precincts, it is possible to refine reconstructed image quality in a manner which is sensitive to a client's spatial region of interest, resolution of interest and components of interest, even without tiling. In fact, it is often preferrable to avoid introducing tile partitions, even when working with huge images with billions of image samples. Using the JPIK protocol and the Kakadu demonstration applications, you can verify such a claim for yourself, since the Kakadu server can efficiently deliver any valid JPEG2000 compressed image whatsoever.

The JPIK protocol is based around the transfer of individual packets from small precincts, each containing quality increments from only a few code-blocks. While all JPEG2000 images must have relatively small code-blocks, not all JPEG2000 images need be partitioned into small precincts. The "kdu-server" deals with this by transcoding[2] all images on the fly into representations involving small precincts. The Kakadu architecture permits such operations to be performed on-demand with very little computational load or memory.

[1] Subbands are outputs of the Discrete Wavelet Transform (DWT). Each subband is a smaller image, corresponding to a sub-sampled band-pass filtered version of the original image.

[2] We use the term "transcoding" to refer to the conversion of one compressed representation into another with the same or similar information. In the present setting, we are referring to the efficient lossless transcoding of an image with large precincts into one with smaller precincts, but the same code-block structure.

1.2 The Importance of Caching

As already noted, precincts are nothing other than containers for code-blocks in a local spatial region of some resolution level. Since the subband synthesis operations performed by the inverse Discrete Wavelet Transform (DWT) are spatially expansive, the image region to which any given precinct contributes overlaps with the image region to which other precincts contribute. While the overlapping of independently coded contributions has the effect of avoiding nasty boundary artefacts such as those found with tiled images, overlapping also means that the amount of information which must be transferred to satisfy a client's region of interest can be significantly larger than one might expect based on the proportion of the entire image occupied by that region.

To avoid excessive inefficiencies, it is essential that the client caches data which is transferred by the server and that the server maintains an understanding of the data which has been cached by the client, so that this data need not be transferred over again. In this way, as the user pans over an image (perhaps the most common user interaction for remote browsing of large images), the effective compression efficiency progressively improves. This, combined with JPEG2000's inherently superior compression performance over JPEG, leads to a convincing scheme for browsing large images remotely over low bandwidth connections. The importance of caching and *"cache mirroring"*[3] by the server was emphasized in the initial reports of the EBCOT algorithm, leading to its adoption in the JPEG2000 standard.

In its current form, the JPIK protocol does not support sophisticated cache models. The client is expected to cache everything delivered by the server and the server is not required to (and generally will not) resend any information which the client is expected to have cached. This is fine for desktop applications, where the memory available to the client often far outstrips the amount of data which the server could transfer over the network in a reasonable period of time. In future versions of the protocol, we will allow the client to specify its cache capacity and the client and server will agree on a policy for discarding inactive elements from the cache.

In future work, we also plan to introduce a mechanism for the client to explicitly inform the server of elements which it has in its cache (the syntax for this should be a trivial re-use of that employed by the server to inform the client of what it is sending). This would enable the client to open a previously browsed image, whose contents have been stored locally, informing the server of the elements which need not be sent again.

1.3 JPIK Protocol Phases

The JPIK protocol consists of three phases:

1. Initial connection is established with the server, which verifies that the requested image file is available and represents a valid JPEG2000 data

[3] Cache mirroring, refers to the fact the activity of the server in tracking the state of the client's compressed data cache.

source (raw code-stream or JP2 compatible file, for now). During this phase, the client and server agree on a new port number to use for ongoing communications, thereby releasing the server's listening port to accept further concurrent client connections.

2. A TCP connection is opened between the client and the server. The server uses this connection to transfer required JPEG2000 headers, including the headers (preamble) of any wrapping file format. The client uses this connection to identify its interests more precisely, which might depend upon the existence of a locally cached copy of some of the compressed data. UUID's or other identifying information may be exchanged during this phase. This so-called "header transfer" phase completes once all headers have been transferred and the client and server have established yet another connection for the purpose of exchanging JPEG2000 packet data and associated acknowledgement messages. By default, this connection is based on UDP datagrams; however, the client and server may agree to use a TCP-only variant of the protocol. Regardless of whether UDP or TCP is to be used for the transport, we refer to this last connection as the *"datagram channel."*

3. The principle phase of a JPIK session involves the transfer of JPEG2000 packets to the client over a TCP-friendly send-acknowledge protocol implemented using the datagram channel. During this phase, the client may use the TCP channel to notify the server of any changes in its spatial region, resolution or image components of interest. These two forms of communication are entirely asynchronous. The server is not obliged to respond immediately (or even at all) to changes in the client's region of interest. The client is obliged to receive and cache packet data transferred by the server, regardless of whether or not this data is immediately relevant to its current region of interest. In practice, the server determines an appropriate point to update its internal notion of the client's region of interest. The server may send data in any order; it may also send data which the client has already cached and acknowledged, either in part or in full. In practice, however, any decent server application will make a serious attempt to mirror the state of the client's cache and only send information which is new to the client. Once the client has all available compressed data for the image, the server is at liberty to disconnect from the client.

As mentioned previously, ongoing data transfers within the JPIK protocol proceed at the level of JPEG2000 packets (a packet is a single quality layer of a single precinct), rather than code-blocks. To ensure fine spatial granularity for these transfers, the Kakadu server transcodes existing compressed data sources on the fly into compressed representations whose precincts are as small as possible. The minimum precinct dimensions are determined by the size of the code-blocks used to compress the image, since the server is unwilling to decode

and re-encode individual code-blocks. One advantage of this transcoding policy is that just about any valid JPEG2000 compressed image can be efficiently served up over a low bandwidth connection. Nevertheless, particular choices of compression parameters can substantially ease the burden on the server's memory and computational resources. For more on this, the reader is referred to the brief discussion appearing in the "Usage-Examples.txt" file.

2 Details of the Protocol

This section provides a detailed description of the JPIK protocol. A description may also be found in the comments at the start of the *"kdu-server.cpp"* file. There have been a number of minor changes in the protocol, within the first couple of weeks of its appearance, so as to fix some obvious oversights and overcome some difficulties encountered with fire walls. For this reason, we note that the present description is backward compatible to Version 3.0.4 (server) and Version 3.0.3 (client) of the Kakadu software. Versions prior to 3.0.3 used a different value for the protocol identification code so as to avoid entirely unexpected results.

In the devlopment which follows, special 16-bit codes are identified by descriptive names. These names are identical to the macros which are used in the implementation of the *kdu-server* application, with the exception that underscores are replaced by dashes in the present text for improved typesetting. The actual values of the codes are provided in Section 2.4 and also in the header file, *"kdu-server-codes.h"*. As an example, the protocol identification code mentioned above is identified here as *KDU-SERVER-PROTOCOL-STD*. When any of these codes must be distributed over the network, they are treated as 16-bit integers and delivered using "big-endian" byte order, also known as "network byte order."

2.1 Connection Establishment Phase

2.1.1 Synopsis

The server listens for new connections on a designated TCP port. The default port number is currently 3771. Clients connecting to this port must provide a 16-bit protocol identification code, the name of the file which they wish to access, and an optional client identification string. In response, the server sends either a 16-bit error code or else three parameters which will define the continued service:

1. a 16-bit quantity indicating the maximum length of the messages or data segments which are used at various points in the communication protocol;
2. the 32-bit IP address for ongoing TCP communications; and
3. the 16-bit port number for ongoing TCP communications.

The server application may opt to provide the ongoing communication service itself (usually in a separate thread or a separate process); or it may delegate this task to another machine or another application running on the same machine.

2.1.2 Detailed Description

Server:

- Listening for TCP connecton, ...
  - The default listening port is 3771, but any port may be used.

Client:

- TCP connect to server's listening port.

Server:

- Accepts client connection and waits for the client to send a 2-byte protocol code.

Client:

- Sends the 2-byte protcol code, *KDU-SERVER-PROTOCOL-STD*, in big-endian byte order.
  - In the future, other client protocols might be accepted.
- Sends a "client identification" string. This might be an empty string; it could contain any information at all, which the client application wishes to send to identify itself in the server log files.
  - The length of the string (number of single-byte characters) is sent as a 2-byte quantity in big-endian byte order.
  - The characters themselves are sent. The string can, but need not contain a null character. If it contains a null character, the server might regard this as the string terminator, ignoring any additional characters sent. In theory, however, there is no requirement that the string contain ascii characters.
- Sends the name of the image file which it wishes to browse. File names are interpreted relative to the server's working directory. The server may choose to restrict access to files which contain drive or directory specifications which access outside the server's working directory. The *"kdu-server"* application offers the *"-restrict"* option for this purpose.
  - The length of the file name string (number of single-byte characters) is sent as a 2-byte quantity in big-endian byte order.

  - The characters themselves are sent. The string can, but need not contain the null-terminator character.

- Waits to receive either an error code or a message length value from the server ...

Server:

- Receives the 2-byte protocol code from the client. Currently, the only protocol code which is recognized for client access is *KDU-SERVER-PROTOCOL-STD*. A separate code, *KDU-SERVER-PROTOCOL-ADMIN* is recognized by the "kdu-server" application for remote administration purposes, but administration is not formally part of the JPIK protocol.
- If the protocol code is not recognized, the server sends the 2-byte error code, *KDU-SERVER-ERROR-PROTCOL*, in big-endian byte order.
- Receives the client identification string.
- Receives the image file name.
- Attempts to open the image file.
  - If the server has already reached its connection limit (too many clients connected simultaneously, or too many sources open simultaneously), it sends the 2-byte error code, *KDU-SERVER-ERROR-TOO-MANY-CLIENTS*, in big-endian byte order.
  - If the file does not exist, the server sends the 2-byte error code, *KDU-SERVER-ERROR-FILE*, in big-endian byte order.
  - If the file exists, but does not have a valid JPEG2000 format (currently, only JP2 files and raw JPEG2000 code-stream files are considered valid), the server sends the 2-byte error code, *KDU-SERVER-ERROR-FORMAT*, in big-endian byte order.
- If any of the above errors occurs, the server terminates the connection; otherwise, it sends a 2-byte quantity, $M_{\max}$, in big-endian byte order. The value of $M_{\max}$ is guaranteed to satisfy $M_{\max} \geq 16$, so as not to conflict with the range reserved for error codes, as described above. The value of $M_{\max}$ represents the maximum length (including the 4-byte header) of the datagram messages used by the server to interactively deliver JPEG2000 compressed data. The client may use this value to pre-allocate buffers of length $M_{\max}$ to receive the datagrams. The same maximum length, $M_{\max}$, applies to other forms of segmented communication, so that the client can use the same set of buffers. Typically, $M_{\max}$, is on the order of about 1000 bytes, but the server may choose to use larger values to trade efficiency for interactivity.

- The server creates a new TCP port for ongoing communications. The "kdu-server" application assigns continuing service to a separate execution thread which uses this new TCP port, thereby freeing up the original listening port to accept new incoming connections. In practice, most server applications should bind the new TCP port prior to sending the message length, $M_{\max}$, as described above. This way, if no more ports are available, the server can send a 2-byte error code, such as *KDU-SERVER-ERROR-CONNECT*, instead of a message length, $M_{\max}$.
- The server proceeds to send both the IP address and the port number on which the service will resume.
  - The IP address is sent first, as a 4-byte quantity in big-endian byte order.
  - The new port number is sent next, as a 2-byte quantity in big-endian byte order.
  - Prior to sending the above information, the server should generally set the new TCP port into the listening mode.
  - The server may delegate the continuing service task to another host or application, sending its IP address and port number instead. Delegation is discussed separately a little later.
- If none of the errors mentioned above have occurred, the server closes the connection establishment channel gracefully and proceeds (in another thread, process or possibly in a delegated host) to wait for the client to call back on the supplied IP address and port number...

Client:

- Receives the message length bound, $M_{\max}$, or error code from the server, processing errors in whatever manner it deems appropriate (the server will have disconnected).
- Receives the IP address and port number for the ongoing TCP communication channel.
- Closes the connection establishment socket gracefully.
- Attempts to connect back on the new IP address and port number supplied for ongoing TCP communications, proceeding with the header transfer phase described in Section 2.2...

Server:

- Accepts the client's back connection to the new IP address and port number, timing out after a reasonable period (e.g., 20 seconds) if no such connection occurs (the client may have been killed in the meantime).
- Proceeds with the header transfer phase described in Section 2.2...

2.1.3 Delegation

As noted briefly above, the server may delegate ongoing service to a separate application which might be running on a separate host. In fact, it is a completely trivial matter to delegate service to a separate instance of the *"kdu-server"* application running on a separate machine or with a separate listening port. The simplest means of doing this (that used by *"kdu-server"* when supplied with a list of delegates) is to open a connection with a potential delegated server, exactly as though it was a client, sending the original client's identification string and file name string to the delegated server. The primary server need not verify that the file exists or that it conforms to a valid JPEG2000 format. Instead, the delegated server does this, returning either a valid message length, IP address and port number, or an error code. The primary server then passes the same information back to the client. When the client re-establishes connection, it will do so on the IP address and port number nominated by the delegated server.

Following the above strategy, it is clear that the primary server does not actually require a local copy of the image files which are being served. In fact, by polling delegated servers one after the other, it is possible to discover the one which does have access to the required file, if any. The *"kdu-server"* application combines this strategy with an algorithm for sharing load between multiple delegates which attempts to minimize the number of different delegates which must be polled in order to establish a successful connection.

2.2 Header Transfer Phase

2.2.1 Synopsis

The header transfer phase of the protocol is conducted exclusively over the ongoing TCP channel which was established at the end of the connection establishment phase. If either the server or the client disconnect from this channel, the session will be terminated.

Once the ongoing TCP channel has been established, the server sends a 16-byte file identification code to the client. While this code is not currently used, it is expected that future enhancements of the protocol will use these 16 bytes to send a UUID or some other identifier, which the client can use to determine whether or not it has a locally cached version of exactly the same image which resides on the server. In this event, the client may be able to skip the header transfer phase. It may even be able to send to the server sufficient information to identify the contents of its cache (e.g., from a previous session) so that the server can intelligently supply only the missing elements. None of this functionality is currently implemented, but it would not be the least bit difficult to add to the current framework. In the meantime, servers should send 16 zero-valued bytes and clients should interpret an all-zero file identifier as unreliable, meaning that the file on the server might have changed since the last time it was served up.

The client sends back a 16-bit code representing the logical OR of a set of option flags. Currently, four options are defined, having the following interpretations:

KDU-SERVER-OPT-WANT-HEADERS The intent of this first option should be clear from the preceding discussion. If the client already has a locally cached version (usually a partial version) of the image, it does not require the headers to be downloaded again. However, since the server will currently only send the unreliable all-zero file identifier, clients should always set this flag. The server will reject options which do not include this flag.

KDU-SERVER-OPT-NO-UDP This flag may be used to request that all UDP traffic be delivered over TCP instead. The interactive communication phase described in Section 2.3 is best conducted over a UDP channel. Unfortunately, many firewalls block UDP traffic so that the TCP workaround is often required.

KDU-SERVER-OPT-WANT-CAPS If this flag is provided, the server will respond by sending a null-terminated sequence of capability descriptors, which the client may use to customize its interaction with the server. See below for more on this.

KDU-SERVER-OPT-NEGOTIATE This flag is always recommended. The behaviour would be mandatory were it not for the fact that older versions of the server and client did not support it. When present, the server responds to the client with a 16-bit code, which is either identical to the client's options code or else a modified version of it. An identical value means that the server supports all the requested options; a zero return code means that the server refuses to continue the session; any other modified code means that the server does not support one or more options, but recommends that the client try again with the modified code. The negotiation loop may continue indefinitely until the parties agree or either part gives up.

If *KDU-SERVER-OPT-WANT-CAPS* was set, the server proceeds to send a sequence of zero or more capability descriptors. Each capability descriptor commences with a 16-bit code, followed by a 16-bit descriptor length value, L. The body of the descriptor contains exactly L bytes and L may not exceed the maximum message length value, $M_{\max}$, which was sent during the connection establishment phase. Currently, only one capability descriptor is formally defined, although clients should generally skip over any descriptors which they do not understand. The descriptor code, *KDU-SERVER-CAP-REGION*, refers to a descriptor whose body consists of 10 bytes, identifying maximum dimensions for the regions of interest which clients can expect the server to correctly serve up. It is legal for clients to specify larger regions, but then the server is at liberty to arbitrarily shrink the region. This gives servers a way of bounding the memory and disk access resources which they can be expected to allocate on behalf of client connections. The 10 data bytes of this descriptor have the following interpretation:

Bytes 0-3: A big-endian quantity, representing an upper bound on the sum of the number of samples in the region of interest, projected onto each image component of interest, at the resolution of interest.

Bytes 4-5: A big-endian quantity, representing an upper bound on the width of the region of interest, projected onto any image component of interest, at the resolution of interest.

Bytes 6-7: A big-endian quantity, representing an upper bound on the height of the region of interest, projected onto any image component of interest, at the resolution of interest.

Bytes 8-9: A big-endian qantity, representing an upper bound on the number of image components of interest.

The server then sends a 16-bit code indicating the number of tiles in the image. This is followed by two blocks of header data. Each block of header data consists of a sequence of data "segments," where each segment is preceded by a 16-bit code indicating the number of bytes in the segment. Segments may contain no more than $M_{\max}$ bytes each, where $M_{\max}$ is the maximum message length value which was sent during the connection establishment phase described in Section 2.1. Each of the two blocks of header data are formed by concatenating its data segments, which are terminated by a zero-valued segment length code.

The first block of header data is intended for communicating optional headers from a wrapping file format. We refer to this block as the "preamble." If the preamble is empty, the client shall interpret the compressed image as a raw JPEG2000 code-stream. In the current implementation of the *"kdu-server"* and *"kdu-show"* applications, a non-empty preamble is used to communicate the header of any JP2-compatible file, from the start of the JP2 signature box, up to (but not including) the SOC marker of the JPEG2000 code-stream in the contiguous code-stream box. The contiguous code-stream box must have a zero-valued length, meaning that the length of the code-stream is unknown a priori. In this way, the preamble header block supports the transmission of colorimetry and other rendering details. There is no particular reason why the header from another wrapping file format could not be transmitted in the preamble.

The second block of header data consists of the main JPEG2000 code-stream header, optionally followed by one or more tile headers. The main header and any tile headers are concatenated into a single data block, whose elements may be separated only by parsing the data in accordance with the specifications outlined in ISO/IEC 15444-1 (JPEG2000 standard, Part 1). The main header commences with the usual SOC marker and contains all the marker segments one would expect to find in a JPEG2000 main header. Tile headers need be sent only for those tiles whose coding parameters differ in some way from those of the main header. In most cases, they will not be required, although it is still legal to send them. Each tile header commences with an SOT marker segment and concludes with an SOD marker, exactly as described in ISO/IEC 15444-1, with the following exceptions:

- Only the tile index in the SOT marker segment is actually important.
- At most one tile header is permitted for each tile and its tile-part index must be 0.
- The SOT fields describing the number of tile-parts for the tile and the length of the current tile-part are currently ignored by the client, but the server would do best to set these to zero for the time being.
- No tile header is followed by any tile data (this is legal, though unusual for a regular JPEG2000 tile-part), since the compressed data is the sent during the interactive communication phase.

It should be noted that all progression order information signalled through the main or tile-part headers is irrelevant from the perspective of the JPIK protocol. Such information may optionally be used if the client later writes the compressed data received to a conforming JPEG2000 file. For this reason, one would not expect to find POC (Progression Order Change) marker segments in the main or tile header components communicated here, although the inclusion of such information would not be illegal.

After delivering both blocks of header data, as described above, the server sends the 16-bit port number to be used in establishing a separate communications channel for the interactive data transfer phase described in Section 2.3. We refer to this separate channel as the "datagram" channel. If the client specified the *KDU-SERVER-OPT-NO-UDP* option, the server listens on this port for the client to complete a TCP connection. Otherwise, the server waits for the client to send back the port number it will be using for the datagram channel, so that a bi-directional UDP connection can be completed. Both the server and client are expected to use the same IP address for the datagram channel as they are using for the ongoing TCP channel.

2.2.2 Detailed Description

Server:

- Sends the 16-byte file identifier. Servers which cannot guarantee that two files with the same identifier will always have identical contents, must send all-zeros.
- Waits to receive a 2-byte option code from the client ...

Client:

- Receives the 16-byte file identifier.
- Sends a 2-byte option code, a numeric quantity, using big-endian byte order. Currently, the option code must be either *KDU-SERVER-OPT-WANT-HEADERS* by itself, or the logical OR of *KDU-SERVER-OPT-WANT-HEADERS* with *KDU-SERVER-OPT-NO-UDP*.

- Waits to receive a 2-byte quantity, indicating the number of tiles in the image ...

Server:

- Receives the 2-byte option code. If the requested options are unrecognized or not supported and the *KDU-SERVER-OPT-NEGOTIATE* option was not included, the server sends the 2-byte error code, *KDU-SERVER-ERROR-FATAL*. If *KDU-SERVER-OPT-NEGOTIATE* was included, the server sends back a 2-byte option code, which is either the same as the client's code or else a minimally modified version of it. If not identical to the client's option code, the server goes back to waiting for a new option code (the option negotiation loop).
- If *KDU-SERVER-OPT-WANT-CAPS* was specified in the options code, the server sends a sequence of zero or more capability descriptors, terminated by the null code (16-bit code, equal to 0).
- Sends a 2-byte quantity, in big-endian byte order, identifying the number of tiles in the image. Note that this cannot be zero; an untiled image has exactly one tile.
- Sends the first block of header data, the "preamble."
  - Sends a sequence of one or more data segments, each consisting of a 2-byte segment length value, in big-endian byte order, followed by the number of bytes identified by the length value. The header block is terminated by a segment with length 0.
- Sends the second block of header data, consisting of the code-stream's main header and any required tile headers. Unlike the preamble, this block of header data cannot be empty.
  - Sends a sequence of two or more data segments, each consisting of a 2-byte segment length value, in big-endian byte order, followed by the number of bytes identified by the length value. The header block is terminated by a segment with length 0.
- Binds a local TCP or UDP port (depending on whether or not the client specified *KDU-SERVER-OPT-NO-UDP*) to be used for the datagram channel. Sends this port number to the client as a 2-byte big-endian quantity.
- If the datagram channel is to use TCP, the server listens on the datagram socket for the client to complete the connection; otherwise, the server waits for the client to send back a 2-byte quantity indicating the port number it will be using for UDP communication over the datagram channel ... If insufficient resources are available to create the datagram channel, the server may send *KDU-SERVER-ERROR-FATAL* instead.

- Enters the interactive communication phase ...

Client:

- If *KDU-SERVER-OPT-NEGOTIATE* was specified, the client receives the 2-byte modified options code back from the server; if it is not identical to the one which the client originally sent, the client must either negotiate a new options code or else it may close the connection.
- If *KDU-SERVER-OPT-WANT-CAPS* was specified, the client processes the null-terminated sequence of capability descriptors returned by the server, discarding any descriptors it does not understand.
- Receives the 2-byte tile count.
- Receives the preamble and code-stream header blocks.
- Receives the port number used by the server for the datagram channel, or *KDU-SERVER-ERROR-FATAL*.
- If the datagram channel is to use TCP (*KDU-SERVER-OPT-NO-UPD* must have been specified), the client attempts to complete the TCP connection back to the server, using its supplied port number; otherwise, the client binds a local UDP port and sends the port number back to the server as a 2-byte big-endian quantity.
- Enters the interactive communication phase ...

2.3 Interactive Data Transfer Phase

2.3.1 Synopsis

The interactive data transfer phase is responsible for the transfer of all of the actual compressed data. This phase involves two separate communication channels carrying two different types of traffic, as follows:

1. The ongoing TCP channel is used by the client to send information regarding its current region of interest. For the present discussion, we shall use the term "Region of Interest" (or ROI) to mean not only the spatial region of interest, but also the maximum resolution of interest and the image components which are of interest to the client. The ROI may change as the remote user zooms or pans a browsing window into the image.

   The server does not use the ongoing TCP channel at all during the interactive communication phase, except to send a 2-byte terminal error code in the event that something went wrong. Both the server and the client are expected to monitor the ongoing TCP channel at all times. If either party disconnects from this channel, the datagram channel is also disassembled and the session terminated. If the client receives any communication from the server on this channel, that communication should be interpreted as a server error code. Error codes which could be

expected from the server include *KDU-SERVER-ERROR-PROTOCOL*, *KDU-SERVER-ERROR-TIMEOUT*, *KDU-SERVER-ERROR-FATAL* and *KDU-SERVER-ERROR-NORMAL*. In most cases, servers will restrict the total amount of time for which clients may remain connected, sending *KDU-SERVER-ERROR-TIMEOUT* once that time expires. The *KDU-SERVER-ERROR-NORMAL* code may be sent if the server finishes transferring all compressed data in the entire image to the client, with the client acknowledging the receipt of such information.

2. The datagram channel is used by the server to send JPEG2000 compressed data. The compressed data is broken into messages which we identify as "datagrams," suggesting UDP as the natural transport for this channel. Nevertheless, as already noted, the client may request that datagrams be delivered using TCP instead of UDP. The datagrams contain their own identification codes which provide sufficient information for the client to insert the compressed data into appropriate locations within its local cache. Although the server will normally send compressed data in an order which is sensitive to the client's current ROI, the client cannot expect this; it should accept and cache whatever data arrives, regardless of its relevance to the current ROI. The server is at liberty to delay processing ROI updates from the client and it will usually be efficient to do so. In fact, the server is even at liberty to entirely ignore all ROI communication from the client.

   The client is expected to send acknowledgement messages back to the server, using the datagram channel. Server datagrams each contain a 4-byte header which is used for the purpose of reconciling delivered and acknowledged messages. Since UDP is an unreliable protocol, the server must implement an appropriate re-transmission policy on the basis of the acknowledgement messages sent by the client. Even when TCP is selected as the transport for the datagram channel, the separate acknowledgement messages are still required. Since TCP is a reliable protcol, re-transmission is not required; however, the acknowledgement messages still provide valuable information which the server may use to ensure that it does not get too far ahead of the client – something which would compromise the server's ability to respond in a timely fashion to changes in the client's ROI.

   It is worth noting that the JPIK protocol specification does not define a server flow control policy. The server is at liberty to use acknowledgement messages in any way it desires to regulate the outgoing data rate and the re-transmission schedule. The *"kdu-server"* application implements one reasonable flow control strategy, which can no doubt be substantially improved.

2.3.2 Compressed Data Encapsulation

As mentioned in Section 1.1, the compressed data in a JPEG2000 code-stream consists of a sequence of "packets" (not network packets). Each packet provides

quality increments for the code-blocks belonging to one image component[4], at one resolution level[5], of one tile (the image might consist of only one tile), within a spatial region known as a "precinct." Interactive communication within the JPIK protocol is based on packets, although packet boundaries are not explicitly identified within the datagrams.

To understand how packet data is encapsulated in JPIK datagrams, it is best to think of the sequence of packets belonging to each precinct as being concatenated into a single precinct data stream. The number of bytes which are transmitted from any precinct's data stream is directly related to the number of packets which are available for that precinct and hence the reconstructed image quality which is available within the relevant spatial region of the tile-component and resolution to which that precinct belongs.

We shall use the term "JPIK increment" to refer to the fundamental unit of compressed data encapsulation within the JPIK protocol. A "JPIK increment" consists of some range of consecutive bytes from some precinct's data stream. Each JPIK increment commences with a header, which uniquely identifies the precinct and the range of bytes from that precinct's data stream which are encapsulated. JPIK datagrams may be composed of one or more whole JPIK increments.

The JPIK increment header consists of the following three quantities, each of which may be regarded as a non-negative integer:

Precinct ID: A unique identifier for the precinct is formed from the zero-based index, T, of the tile to which the precinct belongs, the zero-based index, C, of the image component to which the precinct belongs, and a sequence number, S, according to

$$\text{Precinct ID} = T + (C + S \times \text{num-components}) \times \text{num-tiles}$$

The precincts of any given tile-component are assigned contiguous sequence numbers, S, starting from zero. All precincts of the lowest resolution level (that containing only the LL subband samples) are sequenced first, in raster scan fashion. The precincts of each successive resolution level are sequenced in turn, each in raster scan order within their resolution level.

Range start: The number of bytes in the precinct's data stream prior to the bytes provided by this JPIK increment.

Range length: The number of compressed data bytes provided by this JPIK increment.

[4] A colour image would typically have 3 image components: one for luminance; and one for each of the two chrominance components. However, image components may be used for other purposes than colour components. Alpha and depth channels, layers in a layered document structure, and slices of a three dimensional medical image are some examples.

[5] A resolution level contains those subband samples which are required to double the image resolution available from previous resolution levels. The lowest resolution level consists of the LL (DC) subband, while all other resolution levels consists of exactly three detail subbands, LH, HL and HH, from the relevant level in the DWT decomposition tree.

Each of these three quantities is communicated using a simple byte-oriented extension code. The least significant 7 bits of the first byte in the code hold the most significant 7 bits of the integer quantity being communicated. If these are sufficient, the most significant bit of the byte will be set to 1. Otherwise, another byte follows, whose least significant 7 bits hold the next 7 bits of the integer quantity being communicated. If this byte is still insufficient, its most significant bit will also hold a 0. This continues until sufficient bits have been provided to identify the relevant integer quantity. The last byte of the code, the one which conveys the least significant 7 bits of the integer value being communicated, is the one whose most significant bit is set to 1. This simple extension code is used to communicate each of the three quantities described above, in turn.

The reader may wonder why JPIK increment headers identify byte ranges rather than packet numbers. There are a couple of reasons for this. It is true that servers should generally try to construct JPIK increments from a whole number of packets of the relevant precinct. More specifically, servers should attempt to deliver JPIK increments which represent approximately the same number of packets (and hence quality layers) for each precinct in the client's ROI, before moving on to send additional increments from these precincts. The server may, however, find it convenient and even necessary to occasionally split individual packets into smaller increments. The constraint, $M_{\max}$, on the length of any datagram can force such splitting, because datagrams must consist of a whole number of increments.

An additional reason for using byte ranges rather than packet numbers to delineate JPIK increments is that byte ranges are generally more convenient for client cache implementations, which must be prepared to receive datagrams and hence JPIK increments out of order. If the increments were to arrive out of order and contain only packet numbers, the client would not know how many bytes to set aside in its cache to accommodate the future arrival of the missing packets. While more complex cache architectures could overcome these difficulties, such architectures tend to fragment memory and require additional pointers or data reshu- ing, all of which can become problematic when working with enormous images. To further accommodate the needs of simple and efficient client caching architectures, the following two restrictions apply to the formation of JPIK increments:

1. Regardless of the order in which JPIK increments arrive at the client, the data stream available to the client for any given precinct must never contain "holes" with fewer than 2 consecutive bytes. A hole is a range of missing bytes within the precinct data stream, which is immediately followed by one or more available bytes.

2. Precinct data streams may not contain embedded SOP (Start Of Packet) marker segments. It is perfectly to strip the SOP marker segments from JPEG2000 packets, even if the COD marker segment indicates that SOP marker segments may be used. This is because the JPEG2000 code-stream

syntax can never mandate the use of SOP markers. EPH markers should still appear immediately after each packet's header, whenever the COD marker segment in the main or relevant tile header indicates that their presence is required.

The second restriction, in conjunction with the properties of the MQ coder and the packet header formation rules, as defined in ISO/IEC 15444-1, guarantee that no precinct's data stream will ever contain two consecutive bytes which form a big-endian integer in the range $\texttt{FF93}_\text{h}$ through $\texttt{FFFF}_\text{h}$. The first restriction then ensures that client caches which fill holes in the precinct data stream with $\texttt{FF}_\text{h}$ bytes will always be able to identify the largest contiguous prefix of bytes which is available for the precinct by watching for the appearance of consecutive $\texttt{FF}_\text{h}$'s.

2.3.3 From JPIK Increments to Datagrams

In order to provide uniform quality improvements over the client's ROI, the server generally constructs JPIK increments to hold roughly the same number of new packets from each precinct which contributes to the ROI. This can lead to increments with a wide range of different sizes, some of which may be too small to efficiently transmit as individual datagrams. For this reason, datagrams are allowed to hold multiple JPIK increments. The server generally aims to pack as many JPIK increments as possible into each datagram, without exceeding the maximum message length, $M_{\max}$, communicated during the connection establishment phase described in Section 2.1.

Each datagram contains a 4-byte header. The header is considered part of the datagram so that the body of the datagram may not exceed $M_{\max} - 4$ bytes in length. As we shall see, clients are required to acknowledge the arrival of datagrams, by sending an appropriate message back to the server. The acknowledgement message contains the headers of the datagrams which have arrived. The server then uses the message header to reconcile acknowledgement messages with actual transmitted datagrams.

The server may ascribe any interpretation it likes to datagram headers and generate them accordingly. As an example, the *"kdu-server"* application uses the first 3 bytes of the header to hold a datagram sequence number and the last byte of the header to hold the re-transmission status. In this way, when a datagram is re-transmitted, it is possible to distinguish between the acknowledgement of the original and the re-transmitted datagram. In generating datagram headers, the server is bound only by the requirement that no header may commence with the 2-byte code, $\texttt{0000}_\text{h}$, which is used to distinguish datagrams containing JPIK increments from server notification datagrams (described shortly).

2.3.4 Sending Datagrams over TCP

If the client supplied the *KDU-SERVER-OPT-NO-UDP* option code during the header transfer phase, the datagram channel will use the reliable TCP protocol to send datagrams. In this case, each datagram is delivered to the client by

sending first a 2-byte length code, in big-endian order, and then the actual datagram bytes.

2.3.5 Datagram Acknowledgement Messages

The client uses the datagram channel to send acknowledgement messages back to the server. If the datagram channel is implemented using TCP, the client simply sends the 4-byte header of the acknowledged datagram.

If the datagram channel is implemented using UDP, the client sends a datagram consisting of one or more concatenated datagram headers. Since UDP is an unreliable protocol, the client can and usually should acknowledge each received datagram more than once. The policy implemented by Kakadu's *"kdu-client"* object is to send exactly one acknowledgement message whenever a new datagram arrives. The acknowledgement message contains the headers from the last 3 or 4 acknowledged datagrams, starting from that which arrived most recently.

2.3.6 Region-Done Messages

In addition to JPIK increments, the server may deliver auxiliary notification messages to the client over the datagram channel. All such messages must be conveyed using datagrams whose first two header bytes are both zero – recall that JPIK increments cannot be delivered using datagrams with such headers. The third header byte identifies the particular notification message and the fourth header byte is undefined – servers may write private information into this header byte, which they will receive back when the message is acknowledged. All datagrams sent by the server, including those which contain auxiliary notification messages, must be acknowledged by the client, regardless of whether the client recognizes a particular notification message code or not.

Currently, only one notification message is publically defined, for which the third byte of the header must hold $\texttt{01}_\text{h}$. This message is used to notify the client that the server has sent all available compressed data for some particular ROI. The "region-done" message identifies the ROI for which all compressed data has been sent; it is up to the client to determine whether or not this includes its own current ROI[6].

The body of a "region-done" datagram consists of 7 concatenated 4-byte unsigned integers, each with big-endian byte order. In order, these 7 integers have the following interpretations:

1. The index (starting from 0) of the first image component in the ROI.
2. The number of image components (starting from the one identified above) in the ROI. Of course, a value of 0 here would be meaningless.

[6] Recall that the server need not pay any attention to the client's ROI update messages. In any event, ROI update messages will take time to arrive and the server may not get around to processing them immediately.

3. The number of resolution levels which are discarded from the wavelet decomposition associated with each tile-component in order to arrive at the resolution of interest. A value of 0 means that the entire image resolution is of interest. A value of 1 means that the ROI extends only to half the full image resolution, in the horizontal and vertical directions.

4. The vertical coordinate, y_E of the uppermost edge of the ROI on the code-stream's high resolution grid (or canvas). This is the common grid against which all image component samples are registered and all tile boundaries are defined. The grid forms an absolute coordinate reference, which is independent of the resolution of interest. Values of y_E which lie outside the image region on the canvas will be clipped to that region.

5. The horizontal coordinate, x_E, of the leftmost edge of the ROI on the code-stream's high resolution grid.

6. The height of the ROI on the code-stream's high resolution grid.

7. The width of the ROI on the code-stream's high resolution grid.

2.3.7 ROI Updates

As noted previously, the client may use the ongoing TCP channel to update the server concerning its current ROI. The server is required to clear such messages (prevents clients from getting hung up waiting for the server to clear its queue), but it is not required to respond to them immediately or even at all. The *"kdu-server"* application insists on sending a minimum number of bytes[7] of compressed data for one ROI before moving on to another one. This prevents user interaction with the client browser from interfering with server efficiency.

Client update messages consist of a 2-byte length field, followed by a 2-byte message code and then the bytes of the message body. Both the length field and the message code appear with the usual big-endian byte order. The length communicated via the length field includes both the 2-byte message code and the message body bytes. Currently, only one form of update message is defined, having a message code of *KDU-SERVER-NOTIFY-ROI-RAW* and 28 body bytes, representing the ROI in exactly the same format as that described above for the server's "region-done" messages.

The smallest legal value for the length field of these raw ROI update messages is 30, although larger lengths are permitted. In this case, only the first 30 bytes of the message have a defined interpretation. Servers should be prepared to skip over additional message bytes as well as any messages whose message codes they do not understand, without complaining.

[7] This minimum number of bytes is based upon the server's current estimate of the outgoing transmission rate.

2.4 JPIK Constant Codes

2.4.1 Codes Sent by the Client

Recognized protocol codes

Name	Code
KDU-SERVER-PROTOCOL-STD	5115_h

Recognized option codes

Name	Code
KDU-SERVER-OPT-WANT-HEADERS	0001_h
KDU-SERVER-OPT-NO-UDP	0002_h
KDU-SERVER-OPT-WANT-CAPS	0004_h
KDU-SERVER-OPT-NEGOTIATE	8000_h

Recognized client update message codes

Name	Code
KDU-SERVER-NOTIFY-ROI-RAW	0001_h

2.4.2 Codes Sent by the Server

Recognized terminal error (or normal termination) codes

Name	Code
KDU-SERVER-ERROR-FATAL	0000_h
KDU-SERVER-ERROR-FILE	0002_h
KDU-SERVER-ERROR-FORMAT	0003_h
KDU-SERVER-ERROR-PROTOCOL	0004_h
KDU-SERVER-ERROR-CONNECT	0005_h
KDU-SERVER-ERROR-TIMEOUT	0006_h
KDU-SERVER-ERROR-TOO-MANY-CLIENTS	0007_h
KDU-SERVER-ERROR-NORMAL	$000F_h$

Recognized capability descriptor codes

Name	Code
KDU-SERVER-CAP-REGION	$FFFE_h$

Recognized server notification message codes

Name	Code
KDU-SERVER-NOTIFY-ROI-RAW	0001_h

3 Remote Image Browsing with "kdu-show"

In the future, we will provide additional documentation here. For the moment, however, the reader is requested to refer to the *"Usage-Examples.txt"* file and

to examples on the Kakadu web-site.

4 Image Delivery with "kdu-server"

In the future, we will provide additional documentation here. For the moment, however, the reader is requested to refer to the *"Usage-Examples.txt"* file.