Proposal and Implementation of JPIP (Jpeg2000 Internet Protocol) in Kakadu v3.3

David Taubman, 9 August, 2002

Contents:

1 Introduction

This document describes a protocol for interactive communication of JPEG2000 compressed data, headers and meta-data. The protocol closely follows that proposed by Taubman for the new ISO/IEC/ JTC1/SC29/WG1 work item, known as JPIP (JPEG2000 Internet Protocols), as described in document WG1N2607 and fully implemented within a beta version of Kakadu v3.3. The present document is essentially a duplicate of WG1N2607, with some additions and modifications derived from discussions held at the WG1 meeting in Boston (July 15 to 19, 2002) and incorporated into the first JPIP working draft. Where significant these deviations from the original proposal are identified by red coloured text, within the present document.

The protocol described here has grown out of the original JPIK (JPEG2000 Interactive Kakadu) protocol, for which client and server applications were first distributed with Kakadu version 3.0. For a discussion of the motivation and philosophy lying behind the protocol proposed in this document, the reader is referred to Section 10.

Rather than describing a single protocol, this document actually describes two protocols, which we name “jpip-h” and “jpip-ht”, together with a framework for constructing a family of protocols with similar properties which share common implementation features. The “jpip-h” protocol uses HTTP/1.1 (RFC2616) for all of its signalling. It does not abuse HTTP to tunnel an alternate protocol.

The second variant described here, “jpip-ht”, uses exactly the same request/reply syntax as “jpip-h”, which is again carried over HTTP. However, the actual image data delivered to the client (we call this the *server return data*) is carried on an auxiliary (or slave) TCP channel (the “t” in “jpip-ht” is for TCP). This has the advantage that it allows the server to receive explicit feedback regarding the incremental arrival of chunks of data. This, in turn, enables the server to estimate network conditions and balance the conflicting goals of responsiveness, bandwidth efficiency, and server resource minimization.

While we do not discuss the mechanics here, it is worth pointing out that the communication signalling has been designed for compatibility with additional variants in which the data might be carried over unreliable channels (e.g., UDP). In fact the protocol grew out of one (the JPIK protocol) which was designed specifically for unreliable transports. For some applications, the requirement of a reliable channel with in-order delivery can substantially reduce the performance of client-server communications, since JPEG2000 data can often be used effectively even if it arrives out of order, or if missing elements are abandoned.

Accordingly, we reserve the name “jpip-uu” for a variant of the present protocol family, in which the request/reply messages are carried over UDP, and the server’s return data are carried on a separate UDP channel. For brevity, we shall not consider unreliable variants of the proposed family of protocols any further in this document.

2 Terminology

We use the term *image window* to refer to the spatial region, the image resolution and the image components of interest in a compressed image. Other terms such as “window of interest”, “client window” , “user window” or “server window” are used variously to emphasize the way in which the image window arises. In a graphical user interface, the window might be explicitly

selected by drawing a bounding box on the display. In order to conserve resources, it may be necessary for the server to modify the image window requested by a client, reducing its spatial extent, its resolution, or the number of included image components.

We use the term *standard paragraph* to refer to one or more lines of text, the last of which is a blank line, containing at most white space characters, where each line is delimited by a carriage-return/line-feed (CR-LF) pair. In C/C++, these characters this pair of characters is written "\r\n". HTTP is constructed from collections of such paragraphs, optionally interspersed by length-delimited blocks of binary data and the protocols described here are intended to conform to the HTTP specifications. Each HTTP GET request consists of a single paragraph, while HTTP responses consist of a single paragraph of response headers, optionally followed by response data.

3 Request Syntax

Ongoing client-server communications consists of a sequence of one or more GET requests issued by the client. The server answers each request with a reply and, optionally, by sending some elements of the JPEG2000 compressed image. In this section, we describe only the request syntax.

Requests in both the "jpip-h" and "jpip-ht" variants conform to the HTTP/1.1 specification for GET requests and may be issued indirectly through intermediate HTTP proxies, although this may have an adverse impact on the server's responsiveness. The underlying transport for these requests and the server's replies is TCP, and a persistent TCP connection is desirable, as suggested by the HTTP/1.1 specification. It is legal to close the TCP connection between requests, but this will naturally have an adverse impact on responsiveness and lead to inefficient utilization of network resources. As specified by HTTP/1.1, a client may pipeline its requests, meaning that new requests may be issued before the reply to a previous request has been received.

All requests consist of a single HTTP paragraph, whose first line (the request line) has the form:

```
GET <resource>?<query> HTTP/1.1
```

Here, `<resource>` is any string used by the relevant server to identify the service being requested. According to HTTP recommendations, it should generally commence with a valid URI authority prefix (RFC2396) of the form:

```
http://<host name>/
```

The remainder of the `<resource>` string may specifically identify the image being requested. Alternatively, it may be a server-dependent identifier such as the name of a CGI script. In the latter case, the need may arise to explicitly identify the specific image file through the use of an `Image=<spec>` query field (see below).

The `<query>` string consists of zero or more query fields, each separated by & characters, which are used to specify the client's window of interest (see Section 2 for a definition of *windows*). Fields may optionally be included to identify the image and/or a stateful client-server session. If the server does not understand any component of the query string, it should discard the request and reply with HTTP error code 400 (Bad Request), preferably with an explanatory message to assist in debugging faulty clients or servers.

3.1 Request Query Fields

The following query fields are currently defined.

`R=<spec>`

This mandatory field is used to identify the image resolution associated with the requested window. The `<spec>` string consists of two integers, R_x and R_y, separated by a comma. These integers represent the dimensions of the desired image resolution. In practice, the image is available at only a limited number of resolutions and the server should select the smallest available image resolution, whose actual dimensions, R'_x and R'_y, satisfy $R'_x \geq R_x$ and $R'_y \geq R_y$ if possible. If R'_x differs from R_x, or R'_y from R_y, the server must include a `Resolution:` header in its reply, as described in Section 0.

If R_x and R_y are both equal to 0, the requested window includes no compressed image data and no tile-specific headers, but it does include all mandatory headers. In particular, the main image header (header data-bin 0, as described later) and any elements from a containing JP2-family file format which are essential for meaningful rendering (meta data-bin 0, as described later) are still relevant to the request. The client may issue a request of this form up front in order to recover essential headers before deciding how to proceed in further requests.

`O=<spec>`

This optional field is used to identify the upper left hand corner (offset) of the spatial region associated with the requested window; if not present, the offsets default to 0. The `<spec>` string consists of two integers, P_x and P_y, separated by a comma. The actual displacement of the window region from the upper left hand corner of the image, at the actual resolution selected by the server, is given by $P'_x = P_x R'_x / R_x$ and $P'_y = P_y R'_y / R_y$. Where rounding is required, the server may choose to round up or down. The client should try to form its requests in such a way as to avoid the ambiguities of rounding. Specifically, once the client has sufficient information to know what image resolutions are available, it should use exact values of R'_x and R'_y in the `R=<spec>` field.

`S=<spec>`

This optional field is used to identify the horizontal and vertical extent (size) of the spatial region associated with the requested window; if not present, the region extends to the lower right hand corner of the image. The `<spec>` string consists of two integers, S_x and S_y, separated by a comma. The actual dimensions of the window region, at the actual resolution selected by the server, are computed from $S'_x = S_x R'_x / R_x$ and $S'_y = S_y R'_y / R_y$, with the same rounding considerations as described above for the `O=<spec>` field. The window region need not necessarily be fully contained within the image itself, in which case the server simply takes the intersection between the full image region and the requested window region.

`C=<spec>`

This optional field is used to identify the image components which are to be included in the requested window; if not present, the request is understood to include all available image components. The `<spec>` string consists of a comma-separated list of non-negative integers,

representing the indices of the image components of interest. Image component indices start from 0, and have the interpretation assigned to them by the JPEG2000 code-stream syntax, as described in IS 15444-1. Any non-existent component indices shall be disregarded by the server, but in this event the server must include a `Components:` header line in its reply, indicating the components which are actually being included in the window.

`Cum=<yes|no>`

This optional field may be used to override, or clarify the default policy of replacing any existing window request with the new request. If the value is "yes" (default is "no"), the request is cumulative, meaning that the server is being asked to add the requested window to any existing window, rather than simply replacing the last requested window with the new one. This allows clients to incrementally construct requests for complex image regions. It is possible that servers will not support such a request, in which case they should return HTTP error code 501 (Not Implemented). In this case, the client will need to explicitly sequence requests for the different windows of interest.

`B=<spec>`

This optional field may be used to restrict the amount of data which the server sends in response to this request. If not present, the server should send image data to the client until such point as all relevant data has been sent, a quality limit is reached (see below), or the response is interrupted by the arrival of a new request. There may be little or no reason for using the field with the "jpip-ht" protocol, where the server is able to carefully regulate the flow of response data to the client so as to maintain responsiveness. With the pure-HTTP "jpip-h" variant, however, the server does not receive continuous feedback from the client and may easily push a great deal of data into the pipe, which must be fully received before any data for a new window. To maintain responsiveness with the "jpip-h" protocol, clients should use this field to regulate the flow of traffic and hence maintain responsiveness. Clients will generally need to implement their own flow control algorithms to adjust the request length to changing network conditions.

The `<spec>` string holds a single positive integer, which limits the number of returned data bytes in the bodies of server response messages. As described in Section 5, server responses may be understood as a concatenated list of messages, where each message has a header and a body, containing the actual response data. The lengths of the headers are not counted toward the limit supplied with this field.

`Align=<yes|no>`

This field is optional. The default is "no", meaning that server response data need not be aligned on natural boundaries. If the value is "yes", the combined effect of the server's response messages (if any) to this request must be to augment the client's representation of each data-bin referenced in those response messages to a natural boundary. The natural boundaries for precinct data-bins are whole packets (i.e., layer boundaries), while the only natural boundaries for a code-stream header data-bin is the end of the relevant header. If a prevailing byte limit (`B=<spec>` field) cannot be satisfied without breaking the requested alignment or returning an empty response, the server may exceed the byte limit to the extent necessary.

`L=<spec>`

This optional field may be used to restrict the number of code-stream quality layers which

belong to the requested window. By default, the server should assume that all layers are of interest. The <spec> string holds a single positive integer indicating the number of initial quality layers which are of interest. The server should not attempt to augment any precinct data-bins beyond the relevant layer boundary.

`Image=<spec>`
: This field identifies the image to be served. It will commonly be the image file name, but qualifiers might be added to identify a specific image within a file which is capable of representing multiple images. The field is not required and so may be ignored if the image is already identified by the <resource> string, or by a session ID (`SID=<spec>` field). All non-URI characters in the image file name should be replaced by their hex-hex encoding, as described in RFC2396.

`SID=<Session-ID>`
: This field must be used whenever communication is proceeding within a stateful session, in which the server manages a model of the client's cache. The Session-ID string is returned by the server during session establishment, as discussed in Section 8.

`IID=<Image-ID>`
: This field may be used to supply an Image-ID string, which was previously generated by the server to absolutely identify the image which is being accessed. The image file name may not be unique and does not necessarily correspond to a single encoding of the image content, whereas the Image-ID string should absolutely identify both the image and its encoding. If the image or its encoding has changed, the server should also change the Image-ID string and it should include an `Image-ID:` header line in its reply. See Section 4 for further discussion on the use of the Image-ID.

The request line may be followed by additional headers (one per line, as required by HTTP). If requests are delivered via an intermediate proxy, clients should use HTTP's `Host:` header to specify the name/IP address and optionally the port of the server to which the request is being delivered. However, in accordance with the HTTP/1.1 specification, clients should also identify the target host in the requested resource string. An example of a complete client request is shown below:

```
GET http://www.nice_serve.com/nice.jp2?R=512,640&O=0,0&S=512,478&
      C=0,1,2&Bh=2000 HTTP/1.1
Host: www.nice_serve.com
```

The Kakadu client implementation currently always issues its requests in this complete form, although a client which knows it is directly connected to the server could potentially abbreviate the request by dropping the authority component (prefix) of the resource string and the `Host:` header.

3.2 JPIP Specific Headers

HTTP allows applications to define their own headers, with the understanding that they may not be generically understood. We currently define three request headers which are not already part of the HTTP specification, although there may be a need to define additional headers for communicating client capabilities which intelligent servers might be able to use to customize their responses.

3.2.1 The `Max-Quality:` Header

This header may be used to suggest a limit on the image quality associated with the data which the server sends in response to the window request. Quality limits are difficult to formulate in a reliable manner, and server's are permitted to ignore this header. Nevertheless, it is useful to allow the client to provide some indication of the maximum image quality which might be of interest.

The header takes a single real-valued parameter, *Q*, in the range 0 to 100, which may be interpreted in a manner similar to the ad-hoc Quality-Factor commonly used to control JPEG compression. [Note that this feature is not currently implemented in Kakadu v3.3.]

3.2.2 The `Cache-Contents:` Header

The `Cache-Contents:` header may be used by clients to inform the server of any elements which the client already has in its cache. This can be useful if the client has received some portion of the compressed image during a previous session, or if requests are made outside the context of a stateful session. Rather than have the client explicitly specify the elements of the image which it wants, it is often more efficient to have the client inform the server of the elements it already has. Of course, for efficiency reasons, clients should usually aim to restrict this information to those elements which are relevant to the requested window, but this is not required.

The `Cache-Contents:` header may appear multiple times within a single request paragraph to overcome the common 256-character limit on HTTP header lines. In each occurrence, the header is followed by a comma-separated list of cached element descriptors. Each descriptor identifies either a single data-bin, or a collection of data-bins. Data-bins are discussed more thoroughly in Section 5. For now it is sufficient to know that each data-bin has a unique identifier, within its class, and that there are four data-bin classes: code-stream header data-bins; compressed precinct data-bins; meta data-bins; and catalog data-bins.

Accordingly, each descriptor in a `Cache-Contents:` list commences with one of the characters "H" (header data-bin), "P" (precinct data-bin), "M" (meta data-bin), or "C" (catalog data-bin). The remainder of the descriptor follows one of the following syntactic forms, of which only the first was in the original Kakadu proposal to JPIP (WG1N2607). Each descriptor in the comma-separated `Cache-Contents:` list may employ any of the syntactic forms.

Cache Descriptor Form 1: The class character ("H", "P", "M" or "C") is followed by the unique data-bin identifier which is communicated in the header of each server return message (see Section 5), expressed as a ***decimal*** ASCII string. If the descriptor concludes at this point, the server may assume that the client already has the entire contents of the indicated data-bin in its cache. Alternatively, the descriptor may contain a colon separated suffix, containing a ***decimal*** representation of the number of bytes from the data-bin which the client already has in its cache[1]. As an example of this syntax, a client which already has a JP2 file header (meta data-bin 0), the main code-stream header

[1] It is possible that image data sent by the server in response to previous request arrives after the present request is issued, so the client may actually have a larger prefix of the data-bin in its prefix.

(header data-bin 0), the first tile header (header data-bin 1), all of the first precinct, and the first 267 bytes from the second precinct in the image might include the following header line in its request:

```
Cache-Contents: M0,H0,P0,P1:267
```

Cache Descriptor Form 2: As an alternative to specifying the number of initial bytes available for a precinct data-bin, the number of initial layers (or packets) may be specified instead. This is done by replacing the prefix length (after the colon separator) by the character "L", followed by the number of layers, again as a ***decimal*** value. As an example, the client might indicate that it has the first 7 packets of precinct 3 and the first 11 packets of precinct 12 using the following header line:

```
Cache-Contents: P3:L7,Pb:L11
```

The reader is reminded that this syntax is applicable only when used in conjunction with precinct data-bins.

Cache Descriptor Form 3: The preceding forms may be modified to specify any precinct data-bin explicitly in terms of its individual coordinates as

```
t<tile>c<comp>r<res>p<pos>
```

(note the use of lower case identifiers) instead of

```
P<precinct ID>
```

Here, `<tile>` is the tile index, `<comp>` is the component index, `<res>` is the resolution level, and `<pos>` is the precinct position index within its tile-component-resolution. All indices start from 0 and follow the interpretations described in IS 15444-1. Note that resolution index 0 refers to the lowest resolution level, consisting of the LL subband from the relevant tile-component. In this form, all numeric quantities are expressed in ***decimal***.

Wildcard Generalization: Any of the numerical quantities in any of the above syntactic forms may be replaced by the wildcard character, "*", in which case the cache contents descriptor is automatically expanded to incorporate all elements consistent with the requested window. Some examples are as follows:

- `H*` expands into the main code-stream header and the tile headers of all tiles relevant to the requested window.
- `P*:L2` expands to indicate that the client has the first two packets of all precincts which contribute to the requested window.
- `t*c0r2p*:L10` expands to indicate that the client has the first 10 packets of all precincts from resolution level 2 and image component 0, which are relevant to the request.

It is important to realize that the `Cache-Contents:` header contributes only positive statements concerning the client's cache. The descriptor P30:27 indicates only that the client has at least the first 27 bytes of precinct data-bin 30. The server may know that the client has additional information for this precinct data-bin, either from other `Cache-Contents:`

descriptors, or by keeping track of information it has previously transmitted to the client in the context of a stateful session.

In closing this section, we note that only the first syntactic form given above is completely general. The other forms are supplied to improve the efficiency with which certain types of cache conditions can be signalled. The Kakadu client currently uses the first form alone, without wildcard generalization. Wildcard generalization must be used with special care, since its impact upon the contents of the server's cache model (if any) depends upon the particular image elements which the server considers to belong to the requested window. Clients would do best to use these forms only in a manner which is not sensitive to the server's membership determination or the client's ability to reproduce it. In particular, mixing requests which use wildcard cache contents (or cache needs) statements with others which do not, is not advisable.

3.2.3 The `Cache-Needs:` Header

The `Cache-Needs:` header plays a complementary role to that of the `Cache-Contents:` header. Multiple `Cache-Needs:` lines may be included in any given request, each containing a comma-separated list of cache descriptors, following exactly the same syntax as that described above for the `Cache-Contents:` header. The interpretation of these cache descriptors, however, is different.

If not qualified by a number of bytes or layers, each descriptor in a `Cache-Needs:` header line refers to a data-bin (or data-bins) which are entirely missing from the client's cache. If the server is maintaining a model of the client's cache in a stateful session, it should delete these elements from its model. If the descriptor is qualified by a number of bytes or a number of layers, the server is being informed that these are the only bytes or layers from the data-bin which the client has in its cache for the relevant data-bin (or data-bins). As an example, the following header line informs the server that the client has no information for the first tile header, that it has at most the first 27 bytes of precinct 30, and that it has no information beyond layer 2 (the third layer) for any of the precincts in resolution 2 which are relevant to the requested window:

```
Cache-Needs: H1,P30:27,t*c*r2p*:L2
```

It is important to realize that the `Cache-Needs:` header contributes only negative statements about the state of the client's cache. The descriptor P30:27 indicates only that the client's cache has no information beyond the 27th byte of precinct 30. The server should not infer from this statement alone that the client actually does have any of the first 27 bytes. In a stateful session, information about what the client actually does have may be deduced from previous transactions; otherwise, it may be inferred only from `Cache-Contents:` statements.

Where both `Cache-Needs:` and `Cache-Contents:` headers appear in the same request, they are to be processed in order. For example, if a client wishes to be certain of receiving information only from precinct data-bin 0, starting from the 2nd packet, it may use the following headers:

```
Cache-Contents: M*,C*,H*,P*
Cache-Needs: P0:L0
```

The first header instructs the server that the client has all data-bins relevant to the request window specified in the query string, while the second header instructs the server to delete all layers after

the first (layer 0) from precinct 0 from its model of the client's cache. Note carefully that in a stateful session, the server's model of the client's cache is persistent between requests, so if a new window is subsequently requested, the server will continue to believe that the client already has all data-bins relevant to the first window (there will generally be substantial overlap between windows), unless explicitly informed to the contrary.

Note also that including a data-bin within a `Cache-Needs:` header is not sufficient to ensure that the server will return information about that data-bin in response to the request. The server only returns information which is relevant to the requested window, as specified in the query string. A `Cache-Needs:` descriptor such as `H4` will have no impact on the data returned in response to a request, if the fourth tile does not intersect with the requested window. On the other hand, in a stateful session, header data-bin 4 will be deleted from the server's cache model so that the statement may affect the data returned in response to future requests which do intersect with the requested window. This is a most useful behaviour for clients with limited cache memory, which need to be able to selectively discard elements to make room for more new data.

4 Reply Syntax

The server must issue a reply paragraph in response to each client request paragraph. The reply should conform to the HTTP/1.1 specification, commencing with a status line, which optionally be followed by additional header lines. The status line has the usual form:

```
HTTP/1.1 <status code> <explanation>
```

The `<explanation>` string is arbitrary, but should ideally impart information which is useful for debugging purposes, beyond that conveyed by the status code itself. The following HTTP status codes may be sufficient for many applications, but clients should be prepared to status code which is legal with respect to the HTTP/1.1 specification.

200 (OK)
The server should use this status code if it accepts the window request for processing, possibly with some modifications to the requested window, as indicated by additional headers included in the reply paragraph.

202 (Accepted)
Servers should issue this status code if the window request was acceptable, but a subsequent window request was found in the queue, which rendered the current request irrelevant. This is a common occurrence in practice, since an interactive user may change his/her region of interest multiple times before the server finishes responding to an earlier request, or before the server is prepared to interrupt ongoing processing.

400 (Bad Request)
Servers should issue this status code if the request is incorrectly formatted, or contains an unrecognized field in the query string.

404 (Not Found)
This status code should be issued if the server does not recognize the requested resource or if the Session-ID string supplied in an `SID` query field does cannot be reconciled with an existing session. This may result from unauthorized access attempts or, more likely, from a session time limit expiring. Creating a stateful generally involves the expenditure of server

resources, which must be reclaimed if the client fails to connect to the session within a reasonable period of time.

405 (Method Not Allowed)
Servers might issue this status code if anything other than a GET request arrives, since there is no reason for a JPIP server to understand any of the other HTTP request methods.

The status line is followed by any number of headers (one per line). In most interactive applications, the server should usually include the following cache-control header:

```
Cache-Control: no-cache
```

This prevents any intermediate proxies from caching the response, which generally depends upon the historical context of previous requests made within the same session, or else upon `Cache-Contents:` and `Cache-Needs:` headers, which are not understood by generic HTTP proxies.

A typical server reply might be:

```
HTTP/1.1 200 OK, with modifications
Cache-Control: no-cache
Resolution: 512,512
Offset: 35,130
Size: 221,125
Components: 0
```

Note carefully that the reply may contain extra headers which modify various aspects of the original window request. In fact, the reply should identify all request parameters which are being modified in any way by the server (this should only happen if the 200 status code was issued). The following JPIP specific headers may be included in server replies.

`Resolution:`
The server should send this header if the actual image resolution differs in any way from that requested via the `R=<spec>` query item. The server may need to modify the image resolution because the client requested an image resolution which does not exist.

`Offset:`
The server should send this header if the window region offset differs in any way from that requested via an `O=<spec>` query item. The server may need to modify the offset if it is resizing a requested window region.

`Size:`
The server should send this header if the size of the window region differs in any way from that requested via the `S=<spec>` query item. Most practical servers will likely limit the maximum spatial region for which they are prepared to generate response data, since the window size is directly related to fundamental resource consumption requirements, regardless of the server's implementation architecture.

`Components:`
The server should send this header if the components for which it will serve data are not identical to those requested via any `C=<spec>` query item. Most practical servers will likely limit the maximum number of image components for which they are prepared to generate response data.

`Max-Bytes:`
The server should send this header if the byte limit specified in a `B=<spec>` query item was

too small to be practically achievable, given the server implementation and its resource demands.

`Max-Layers:`
The server should send this header if the layer limit specified in an `L=<spec>` query item was larger than the actual number of quality layers available from the code-stream.

`Image-ID:`
The server should send this header if the server's unique image identifier differs in any way from the identifier supplied with an `IID` field in the query string. The Image-ID is an arbitrary, server-assigned string, except that it must consist only of legal URI characters (see RFC2396) and all ASCII representations of the value 0 are reserved. If the query string contains the field `IID=0`, the server is obliged to include an `Image-ID:` header line indicating the actual Image-ID. If the server's reply includes the header line,

```
Image-ID: 0
```

the server is unable to assign unique image identifiers and hence validate the integrity of an image between multiple requests or sessions. Whenever the server supplies an Image-ID which is different from that in the query string, it should also disregard all `Cache-Contents:` and `Cache-Needs:` headers in the request, so that only the window request itself has any meaning.

In the "jpip-h" protocol variant, the response data itself is sent as an HTTP entity body following the reply paragraph. In this case, the reply paragraph should also include a `Content-Type:` header, plus either a `Content-Length:` header or a `Transfer-Encoding:` header, specifying HTTP's chunked transfer coding. These matters are discussed further in Section 6, although we note here that the content type (i.e., the MIME type) for self-describing JPIP response data is "image/jpip-stream".

In the "jpip-ht" protocol variant, the server's HTTP reply paragraphs should not be followed by entity bodies. Instead, the response data should be sent over the auxiliary TCP channel.

4.1 On Server Modification of Client Requests

As explained in Section 10, the nature of JPEG2000 images is such that a resource constrained server cannot be expected to satisfy any arbitrary request, exactly as phrased by the client. Rather than simply rejecting such requests, which would be of little assistance to the client, the server is allowed to respond to a modified request, so long as it informs the client of the particular parameters which are being modified. As indicated above, this is done by the inclusion of any relevant headers in the reply paragraph.

Although modifications are allowed, the development of delinquent servers should be prevented by placing constraints on the way in which requests may legitimately be modified. Specifically, server modifications never increase the scope of a request, except to increase the resolution to the lowest resolution supported by the underlying image. Servers should only restrict the scope of a request in order to limit resource consumption so as to avoid unfairly penalizing other clients. Moreover, this should be done in a systematic way, so that a client which limits its request in the manner suggested by the server's reply to an earlier request, should find that the new request is not modified by the server.

5 Response Data Syntax

The server response to any given request consists of a concatenated sequence of *messages*, each of which represents an incremental contribution to a single data-bin. Each message is completely self-describing, so that the sequence of messages may be terminated at any point. To maintain responsiveness, servers will generally pick appropriate points at which to terminate a message sequence after receiving a new request indicating a different window of interest. In this section, we are concerned with the contents of the syntax and interpretation of the server response messages, rather than implementation strategies for servers. We will see in Sections 6 and 7 how the sequence of response messages is actually transported in particular transport environments.

5.1 Message Structure

Each message represents an incremental contribution from exactly one data-bin (see next sub-section). The message header consists of the following elements (in order): 1) a unique data-bin identifier; 2) a byte offset from the start of the data-bin to the first of data provided by this message; and 3) the number of data bytes provided by this message. All three quantities are variable length encoded.

Each of these three header elements consists of a sequence of bytes, terminated by the first byte whose most significant bit is 0. The integer-valued offset and length quantities (items 2 and 3 above) are formed by concatenating the least significant 7 bits of each byte in its code, in big-endian order. The data-bin identifier is somewhat more complex; it is described in the next sub-section.

5.2 Data-Bin Types and Identifiers

Data-bin identifiers must be able to uniquely identify any given precinct, image header, or metadata entity in the image. Each identifier consists of two components: a) a variable length class identification code; and b) a unique identifier within the relevant class. These components are packed into a single variable-length byte code as follows.

The code consists of *k* bytes, the first *k-1* of which have an MSB of 1, the final byte having an MSB of 0. Conceptually, the least significant 7 bits of each byte in the code are concatenated in big-endian order to form a single integer identifier. The class identification information appears as a variable length binary code in the most significant bit positions of this integer, while the in-class identifier is the integer formed by stripping away the bits used by the class code.

Our current proposal involves four distinct data-bin classes: precinct data-bins; code-stream header data-bins; meta data-bins; and catalog data-bins. Each of these classes has a pair of class identification codes, where the second code in each pair carries the extra information that the relevant data-bin is completed by the contribution in this response message.

- Length 1, code="1": **precinct** data-bin
- Length 2, code= "01": completed **precinct** data-bin
- Length 3, code="100": **header** data-bin
- Length 4, code="1000": completed **header** data-bin
- Length 5, code="10000": **meta** data-bin
- Length 6, code="100000": completed **meta** data-bin

- Length 7, code="1000000": **catalog** data-bin
- Length 8, code="10000000": completed **catalog** data-bin

From the above, one may conclude that no valid class identifier may commence with the 0 byte, which is reserved for use as a terminator, as discussed in Section 5.4.

5.2.1 Precinct Data-Bins

For data-bins belonging to the precinct class, the proposed in-class identifier is formed exactly as in the JPIK protocol (WG1N2392). Specifically, the identifier, *I*, is given by

$$I = t + (c + s \times \text{num\_components}) \times \text{num\_tiles}$$

where *t* is the index (starting from 0) of the tile to which the precinct belongs, *c* is the index (starting from 0) of the image component to which the precinct belongs, and *s* is a sequence number which identifies the precinct within its tile-component. Within each tile-component, precincts are assigned contiguous sequence numbers, *s*, as follows. All precincts of the lowest resolution level (that containing only the LL subband samples) are sequenced first, starting from 0, following a raster-scan order. The precincts from each successive resolution level are sequenced in turn, again following a raster-scan order within their resolution level.

It follows that a precinct identifier of 0 refers to the upper left hand precinct from the LL subband of image component 0 in tile 0.

5.2.2 Header Data-Bins

For data-bins belonging to the code-stream header class, the proposed in-class identifiers have the following meaning:

- A value of 0 refers to the code-stream's main header. This data-bin consists of a concatenated list of all markers and marker segments in the main header, starting from the SOC marker. It contains no SOT, SOD or EOC markers.
- A value of $n \geq 1$ refers to the tile header for tile $n-1$. This data-bin consists of a concatenated list of all markers and marker segments in the initial tile-part header for tile $n-1$. It does not contain the SOT marker segment or the SOD marker.

Note that there is no reason for the protocol to be aware of tile-parts or to transfer non-initial tile-part headers, since precincts are addressed directly, regardless of how their constituent packets may be distributed amongst tile-parts in a JPEG2000 code-stream. Tile-parts serve no role in JPEG2000 other than that of sequencing packets in a static code-stream, so there should be no need for them in an interactive protocol.

5.2.3 Meta Data-Bins

[Note carefully that the material in this section is only a proposal. Some additional details may need to be fleshed out, and we have only implemented transport for meta data-bin 0, as defined below.]

Meta data includes all information which is related to the image, but not strictly required to perform the basic JPEG2000 decoding steps. In practice, the meta data is likely to be drawn from the boxes of a JP2-family file which contains the code-stream of interest. However, one can easily imagine circumstances in which the server might generate some meta data elements dynamically.

Each meta data-bin consists of one or more complete JP2-family boxes. Nesting properties of original boxes shall also be preserved. No box may appear in more than one data-bin, but the server is allowed to decide how boxes should be partitioned into data-bins, subject only to the restriction that the same file (as identified by any UUID sent during connection establishment) shall always be served using exactly the same box partition.

The current proposal does not assign any particular interpretation to the data-bin identifiers themselves, with the exception of the data-bin with identifier 0. The interpretation of all other meta data-bin identifiers is to be conveyed by catalog data-bins, if required. The special meta data-bin, having identifier 0, is defined to hold all *essential* boxes from any JP2-family file which contains the code-stream. *Essential* boxes are those which are required for a client to correctly render the image. The server may choose to include additional boxes into this data-bin if it wishes, but in accordance with the restrictions mentioned above, the assignment of boxes to meta data-bin 0 must be consistent across client connections. For JP2 files, meta data-bin 0 should include the JPEG2000 signature box, the file type box and the JP2 header box, with all of its associated sub-boxes.

Although we currently place no restriction on the way in which meta data-bins identifiers are constructed by the server, we note that the data-bin headers will be smallest if the identifiers are drawn from positive integers, densely clustered around 0. This may also maximize the efficiency of certain client cache implementations – it would help the Kakadu client cache implementation to use as little memory as possible.

5.2.4 Catalog Data-Bins

[Note carefully that the material in this section is only a proposal. Some additional details will need to be fleshed out.]

A client which receives one or more meta data-bins, may parse their contents to determine the particular JP2 boxes which it has. However, in some cases the interpretation of those boxes may depend on their positional relationship to boxes which might be stored in other data-bins. For this reason, auxiliary information may need to be sent to identify the organization of boxes. This auxiliary information, represented by catalog data-bins, is also of interest to clients wishing to learn about the structure of the available meta-data so as to make informed queries for specific meta data-bins of interest.

Since we have no current implementation of catalog data-bins, the information presented here is deliberately sketchy. What we have in mind is an hierarchical structure of catalog data-bins, rooted at the special catalog data-bin with identifier 0. The assignment of identifiers to all other data-bins is left up to the server, so long as it is consistent across all sessions over which the same image is to be served.

Each catalog data-bin contains a concatenated list of box segments, where each segment commences with the 4-byte box signature and concludes with a segment description – there is no

box length. Each segment's description can take one of four forms: 1) a list of sub-boxes, each with their own descriptions; 2) the identifier of another catalog data-bin, which expands the contents of the box; 3) the identifier of a meta data-bin, together with a byte range (offset and length) within that data-bin, at which the entire contents of the box may be found; or 4) a combination of the second and third forms.

We do not go so far here as to describe specific signalling for the various box description possibilities mentioned above, although any number of schemes could obviously be used.

5.3 Server Restrictions

It is generally undesirable to impose too many restrictions on the way the server may construct its response messages, firstly since this may obstruct good server implementations and secondly since it would make the protocol description larger and more difficult to comprehend. For this reason, we allow servers to construct messages in any manner whatsoever, subject only to the following requirement:

> Within any connected session, the server must guarantee that the data which the client receives for any given data-bin must not contain any non-terminal missing sections ("holes") which are smaller than 8 bytes in length.

This restriction enables clients to efficiently slot the received data into variable length structures without incurring a memory overhead for managing holes, which might arise due to misordering of the messages in the network or elsewhere. There is no reason why a server cannot adhere to this convention. For most envisaged server implementations, it is sufficient simply to restrict the message body to at least 8 bytes, except for a data-bin's terminal message, which may have any length.

Since we do not impose any other restrictions on the order in which the server may send data, it is possible that compressed precinct data arrives before the code-stream header information required to decode it correctly. It is the client's job to handle such situations. The client should not assume that a tile header uses the same coding parameters as the main header, until it has received a message from the server which explicitly identifies the contents of the tile header. This message will commonly indicate that the tile header data-bin is empty, but it must be sent before the client can be sure that it is able to correctly decode data for the tile.

5.4 End of Response (EOR) Message

We reserve the single byte identifier, 0x00, for the special "end-of-response" message. In the "jpip-ht" variant, response data is sent on a separate TCP channel to the server's reply text. In this case, it is essential that the response data for each client request be explicitly terminated with an EOR message, since otherwise the client could not determine the correspondence between requests and responses. Establishment of this correspondence is particularly important if the client requests manipulate the server's state through use of the `Dw` or `Dp` query items (see Section 3.1).

For the "jpip-h" variant, the end of the HTTP response data is already clearly identified. Nevertheless, the inclusion of a terminal EOR message in the response is still required, since it carries extra information regarding the reason for termination of the response.

A complete EOR message consists of the single byte identifier, 0x00, followed by single-byte reason code, *R*, and then a byte count (variable length encoded in the usual way), indicating the number of bytes in the body of the EOR message. We do not propose here any particular interpretation for the EOR message body and we do not currently intend include any body with the EOR messages sent by the Kakadu JPIP server; however, this mechanism opens the door to possible extensions. It could also be used as a means of padding responses to specific length boundaries. We note that the EOR message body does not contribute to the byte count restricted by any `B=<spec>` query item (see Section 3.1).

The following reason codes are currently defined:

R=1 (Image Done)
: The server has transferred all available image information (not just information relevant to the requested window) to the client.

R=2 (Window Done)
: The server has transferred all available information which is relevant to the requested window.

R=3 (Window Change)
: The server is terminating its response in order to service a new window request which has arrived.

R=4 (Byte Limit Reached)
: The server is terminating its response because the byte limit specified in a `B=<spec>` query item (see Section 3.1) has been reached.

R=5 (Quality Limit Reached)
: The server is terminating its response because the quality limit specified in a `Max-Quality` request header (see Section 3.2.1) has been reached.

R=6 (Time Limit Reached)
: The server is terminating its response because some time limit has been reached.

6 Description of the JPIP-H Interactive Protocol

As already mentioned, "jpip-h" protocol is constructed entirely from a small subset of the HTTP/1.1 specification. All client requests are HTTP GET requests, following the syntax set out in Section 3. In response to each request, the server sends an HTTP reply paragraph, optionally followed by an HTTP entity body. The structure and interpretation of the reply paragraph are described in Section 4. If the reply is followed by an entity body, the entity data is a sequence of response messages, following the syntax described in Section 5.

The response data attached to any given server reply must conclude with exactly one EOR message (see Section 5.4). The server should generally use the *chunked* transfer encoding defined by HTTP/1.1 to transfer an entity body. In fact, RFC2616 mandates the use of the chunked transfer encoding for all entity bodies whose length cannot be determined at the point when the

reply paragraph is written. With this transfer coding, the reply paragraph should include the header line

```
Transfer-Encoding: chunked
```

The reply paragraph is followed by an alternating sequence of chunk headers and chunk data. The chunk header consists of a single line of text, recording the number of bytes of chunk data which will follow, as a ***hexadecimal*** string. Each block of chunk data is followed by a blank line. The response is terminated by a response having zero length, so all non-terminal chunks must have non-zero length. For a comprehensive description of the chunked transfer-encoding, refer to RFC2616.

Although the Kakadu server does construct each of its chunks from a whole number of response messages, this is not a requirement, and clients should not make any assumptions about the chunk boundaries. This is because intermediate proxies are at liberty to change the chunk boundaries, or even to remove the chunked encoding.

If the HTTP's chunked transfer encoding is not used, and the server intends to include an entity body, the full length of the concatenated sequence of response messages which the server intends to send must be identified by a `Content-Length:` header in the reply paragraph. Obviously, this is less desirable, since there is no way for the server to pre-empt a partially transmitted response if new window requests arrive.

We conclude this section by providing the initial set of client request and server response messages from a real communication session, using the Kakadu implementation of "jpip-h." Each line of communications from the client to the server are identified by the → symbol, while each line of communications from the server to the client are identified by the ← symbol. Note that new client requests are issued before server response data is complete. Clients should generally pipeline requests in this way, in order to minimize the impact of round-trip delays on bandwidth efficiency.

```
→GET
      http://169.254.237.107/jpip-h?SID=033C38BE485819B9&R=0,0&B=
      2000 HTTP/1.1
→Host: 169.254.237.107
→
    ←HTTP/1.1 200 OK, with modifications
    ←Resolution: 64,80
    ←Size: 64,80
    ←Components: 0,1,2
    ←Cache-Control: no-cache
    ←Transfer-Encoding: chunked
    ←Content-Type: image/jpip-stream
    ←
    ←400
    ←<1024 bytes of chunk data>
    ←
→GET http://169.254.237.107/jpip-h?SID=033C38BE485819B9&R=64,80&O
      =0,0&S=64,80&C=0,1,2&B=2000 HTTP/1.1
→Host: 169.254.237.107
→
    ←3e2
    ←<1000 bytes of chunk data>
    ←
```

```
     ← 0
     ←
     ←HTTP/1.1 200 OK
     ←Cache-Control: no-cache
     ←Transfer-Encoding: chunked
     ←Content-Type: image/jpip-stream
     ←
     ←400
     ←<1024 bytes of chunk data>
     ←
→GET http://169.254.237.107/jpip-h?SID=033C38BE485819B9&R=512,640
      &O=0,0&S=512,478&C=0,1,2&B=2000 HTTP/1.1
→Host: 169.254.237.107
→
     ←3e1
     ←<993 bytes of chunk data>
     ←
     ←0
     ←
     ←HTTP/1.1 200 OK
     ←Cache-Control: no-cache
     ←Transfer-Encoding: chunked
     ←Content-Type: image/jpip-stream
     ←
...
```

7 Description of the JPIP-HT Interactive Protocol

The “jpip-ht” protocol is identical to “jpip-h”, except that the server return data is delivered over a separate TCP channel. To complete its connection with the server, the client must open a TCP connection to the host and port numbers supplied during session establishment (see Section 8), and send a single HTTP-like request paragraph, of the form

```
CONNECT <resource>?SID=<Session-ID> JPIP-HT/v1.0
```

where `<resource>` is the string returned via the `Session-Resource:` header described in Section 8.2 and `<Session-ID>` is the string returned via the `Session-ID:` header, also described in Section 8.2.

As with HTTP paragraphs, each line must be terminated by a carriage return/line-feed pair, and the last line must be empty. The connection paragraph may include any number of extra header lines, but no headers are currently defined. The server acknowledges the connection request by sending a reply paragraph of the form

```
JPIP-HT/v1.0 200 OK
```

If an error occurs, the server may send one of the HTTP error codes on the status line of this reply paragraph. Again, extra header lines may be included, but no headers are currently defined.

Server return data is sent on the auxiliary TCP channel as a sequence of length-delimited data chunks. All such communication is binary. The first 2 bytes of each chunk hold a 2 byte big-endian integer, which identifies the length of the chunk. The length identifier itself is included in the chunk length, so lengths may not be less than 2, even if no actual data is included in the chunk. The server is free to determine appropriate chunking boundaries.

There are two key distinctions between the way in which response data is constructed and used in “jpip-ht” and “jpip-h”. Firstly, receipt of each chunk must be explicitly acknowledged by the client. Once a chunk has been completely received, the client sends a 4-byte identifier back to the server using the auxiliary TCP channel to do so. We may later provide a definition for the contents of this 4-byte identifier, but for now clients should send 4 zeros, which will always be a safe option. The acknowledgement identifier should be used by the server to regulate the flow of data to the client in such a way as to maintain responsiveness while fully utilizing the available communication bandwidth.

The second distinction between “jpip-h” and “jpip-ht” is that each request sent on the HTTP channel must be matched by an EOR (End of Response) message in the response stream, regardless of whether or not the server’s reply is accompanied by any response data. The client has no other way of associating request/reply pairs with response data. Recall that in “jpip-h”, the server includes an EOR message only when an entity body is appended to its reply. In “jpip-ht”, however, the server must send an EOR message for each request to which it replies with a 200 series status code, including the 202 status code. In the latter case, the EOR reason code 3 (Window Change) should be used, as explained in Section 5.4. There is no requirement that responses to distinct requests be packed into distinct return chunks. In fact, a single chunk may contain multiple EOR messages and this can be a useful mechanism for delivering the EOR messages for one or more pre-empted requests before sending the response data for the most recent window request (the one which pre-empted earlier requests).

8 Session Establishment

We have already introduced the concept of a stateful session between the server and the client. Within the context of a single session, the server maintains a model of the client’s cache, which is augmented whenever new image information is sent to the client or when client requests include `Cache-Contents:` headers. Elements may also be discarded from the model when client requests include `Cache-Needs:` headers. Sessions are currently required for the “jpip-ht” protocol, while the “jpip-h” protocol may be used without sessions if desired, in which case every request must be entirely self-contained.

In this section, we describe the currently implemented procedure for establishing a session between the client and the server. At a minimum, session establishment consists of a client request for a particular image, in response to which it expects to receive details concerning the resource which must be queried and the Session-ID string which must be supplied when making requests within the session. It is conceivable that these session establishment parameters might be obtained via e-mail exchanges, or even ftp transactions.

In the present document, we describe session establishment via HTTP. This allows HTTP’s security and authentication services to be leveraged to impart appropriate levels of security to the ongoing image communications. If necessary, session establishment may involve the exchange of encryption keys. In any event, the security of the ongoing communications can be arranged to depend entirely on the security of the initial session establishment communication.

8.1 Session Establishment Requests

The client sends an HTTP GET request to a session establishment server, identifying only two pieces of information: 1) the requested image file; and 2) the requested protocol variant. The protocol variants which are supported by the client are identified as acceptable MIME types in the GET request. In particular, the following MIME types are proposed:

session/jpip-h
Refers to the session parameters for a "jpip-h" session.

session/jpip-ht
Refers to the session parameters for a "jpip-ht" session.

A typical HTTP session establishment request might be as follows:

```
GET http://www.exposed.host/very-nice-image.jp2 HTTP/1.1
Host: www.exposed.host
Accept: session/jpip-ht
```

As an alternative to specifying the desired image in the request's resource string, the image may be identified in a query string, using the `Image=<spec>` field described in Section 3.1. This allows requests to be made through a CGI script. It also allows the protocol variant to be identified explicitly or implicitly through the resource name. For example, the above session establishment request might be rephrased as:

```
GET http://www.exposed.host/jpip-ht.cgi?Image=very-nice-image.jp2
      HTTP/1.1
Host: www.exposed.host
```

In this case, the inclusion of an `Accept:` header is optional. In any event, the server is free to define acceptable request syntax in any way it likes.

8.2 Session Establishment Responses

In our HTTP implementation, the session parameters are encapsulated within an entity body, which is attached to the server's reply. The protocol variant to be used within the session is supplied in the reply paragraph's `Content_Type:` header. An example is shown below:

```
HTTP/1.1 200 OK
Server: Kakadu JPIP Server v3.3
Cache-Control: no-cache
Content-Type: session/jpip-ht
Content-Length: 133

Session-Host: 169.254.118.4:80
Session-Resource: jpip-ht
Session-ID: 033C38BE485819B9
Image-ID: C6D5D8283C479878ED14C3BE49B8FA2A
```

Note that the entity body follows the reply paragraph. Its length is indicated by the usual HTTP `Content-Length:` header. An existing web browser should be able to deliver the entity body text to an application (or plug-in) which is registered against the relevant MIME type ("session/jpip-ht" or "session/jpip-h") if necessary. The entity itself is nothing but a sequence of CRLF delimited lines of plain ASCII text, each of which contains a single header. The following headers are defined:

`Session-Resource:`

Identifies the resource name to be used in constructing all requests. This header is mandatory.

`Session-ID:`
: Identifies a string token to be supplied with the `SID` field in the query component of all window requests within the session. This header is also currently mandatory, although it is conceivable that the session could be identified by the resource name alone.

`Session-Host:`
: Holds the network naming authority host name, or IP address of the image server, to be used in on-going session communications. In many cases, the actual image server may have a network assigned IP address, not having a public life. This enables the session establishment server to employ load balancing algorithms to distribute session requests to any number of unregistered machines. The name or IP address may optionally be followed by a port number, separated by the usual colon delimiter, as shown in the example above. Port 80 is the default, which makes its appearance in the above example redundant. For the "jpip-ht" protocol, a second port number may be supplied for the auxiliary TCP channel. If present, the second port number is separated from the first by a comma. If not, the server listens for both the main HTTP communication channel and the auxiliary TCP channel on the same port.

`Image-ID:`
: The string supplied with this header has the same interpretation as the string supplied with the `IID` query field (see Section 3.1) and that returned via the `Image-ID:` reply header (see Section 4). The header is optional, but clients should not expect to be able to re-use information cached from a previous session with an image of the same name, unless an Image-ID was issued and can be correctly matched up. In the context of a session, there is no need for requests to include the Image-ID in an IID field, since the Image-ID must remain fixed throughout the session.

If the address and port details of the session host happen to coincide with those of the session establishment server, the client may re-use the existing open TCP connection over which it received the session establishment response to issue all of its session requests. This avoids the TCP tear-down and re-connection overheads. In practice, however, load balancing session establishment servers may frequently delegate requests to different image servers.

9 Non-Interactive Communications

Although we have hitherto described protocol(s) for interactive image communications, it is easy to see how these mechanisms can be used for non-interactive access to a specific window of interest within a JPEG2000 image. Consider, for example, the following HTTP request paragraph:

```
GET http://www.exposed.host/nice.jp2?R=512,640!O=0,0!S=512,640!C=
      0,1,2,3 HTTP/1.1
Host: www.exposed.host
Accept: image/jpip-stream
```

The server's response should be identical to that which would be issued within a "jpip-h" session. For example, the response to the above request might be as follows.

```
HTTP/1.1 200 OK, with modifications
Resolution: 700,800
Size: 700,800
Components: 0,1,2
```

```
Transfer-Encoding: chunked
Content-Type: image/jpip-stream

400
<1024 bytes of chunk data>

3e2
<1000 bytes of chunk data>

0
```

Note that non-interactive server responses can probably be reliably cached by intermediate proxies. For this reason, the server will not normally include a `Cache-Control:` header in this case.

Image requests following this syntax could easily be embedded in HTML text. A web browser issuing the request would be able to direct the response data to an appropriate application (or plug-in), based on its indicated MIME type, "image/jpip-stream". More generally, the string of server messages which constitute an "image/jpip-stream" type response are fully self-describing. They can be shipped around a network in whole or in part, in order or out of order, regardless of how they may have been requested, if requests were involved at all.

Clients which need or can support response data in an alternate form, may include additional MIME types within the `Accept:` header. For example, the request might be issued as

```
GET http://www.exposed.host/nice.jp2?R=512,640!O=0,0!S=512,640!C=
      0,1,2,3 HTTP/1.1
Host: www.exposed.host
Accept: image/jpip-stream, image/jpeg, image/jp2
```

If a complete image (JP2, JPEG, etc.) is returned in place of a string of jpip-stream messages, the returned image can still be correctly positioned on a browser's window by comparing the requested window parameters with the modified values returned via any `Resolution:`, `Offset:` and `Size:` headers in the reply paragraph.

10 Philosophy (a brief discourse)

The protocols described in this document have evolved from the JPIK protocol (WG1N2392), which is based on the following two philosophical principles.

1. All image data returned by the server should be completely self-describing, meaning that its interpretation does not depend upon the sequence or content of any requests made by the client.
2. The client's requests should indicate what the user is actually interested in seeing, rather than the JPEG2000 compressed image elements which the client thinks it should ask for.
3. The server should be permitted to modify client requests which impose excessive resource requirements.

The first of these principles has a number of important implications. It ensures that the return data can be common to a wide variety of request languages and applications. In Section 9, for example, we see that non-interactive retrieval of an image window by means of a simple URI which can be embedded in an HTML document, or written on the back of an envelope, results in the same type of response as an interactive request. As another example, a motion JPEG2000

source might be streamed to a remote client using exactly the same return data syntax as that described here for interactive image retrieval. In yet another example, a client which is involved in an interactive communication session with a server could stream the responses which it receives directly to another application or client, which is not involved in communication with the server.

In addition to the ability to create new applications which use the same return data format, a second advantage which stems from our first philosophical principle is that the return data can be received out of order. So long as the integrity of individual response messages is preserved (these can be arbitrarily small), the order in which the messages arrives is of no consequence. An application which receives the first and third messages can use their contents to improve the available image quality, without waiting for the second message to arrive. This allows the protocol to be deployed over unreliable transports such as UDP, which can have significant benefits if the transport is subject to significant loss. If an interactive user's window of interest changes, there is no need for the server to resend lost packets whose contents are no longer relevant. In high loss environments, TCP's ordered delivery policy can hamper the timely delivery of the image information required by an interactive user.

The second principle stated above, is clearly embodied in the image-oriented request syntax described in Section 3. Each request explicitly informs the server of the particular image region, resolution and image components which are currently of interest to the client. This enables the server to fetch all of the relevant image elements from disk using as few accesses as possible, minimizing the risk of disk thrashing as multiple clients compete for the server's limited resources. The Kakadu server pre-fetches and assembles a prescribed amount of data in response to each new window request, where this amount of data is determined dynamically based on network conditions, so as to maintain responsiveness to clients on low bandwidth connections, while preventing clients on high bandwidth connections from usurping the server's resources.

One can envisage applications in which the server must dynamically create the compressed image data in response to interactive client requests. Again, the efficiency of this process is likely to be maximized if the server is explicitly informed of the spatial region(s), resolution(s) and image component(s) in which the user is interested, as dictated by our second philosophical principle.

It could be argued that intelligent clients should be able to make their own decisions regarding the compressed image elements which should be delivered and the order in which they should arrive. While this has the appearance of giving intelligent clients the control they need, it should be noted that before an intelligent client can make an "intelligent" request for specific image elements (as opposed to generic image regions and resolutions), it must first determine the meaning and the size of each of these elements, and this can involve the exchange of significant additional information. Allowing clients to make microscopic requests for arbitrary image elements, in any order they choose, could also damage the server's ability to respond to other clients.

Since there are generally many more clients than servers, it is easier to upgrade servers than clients. This means that application-tuned servers, or simply smarter server implementations are more likely to be deployed rapidly than specialized clients. This adds weight to the argument that the server generally knows best and should be allowed to come up with the best serving policy, being informed of the end user's actual interests.

As an example of the benefits of freeing the server to decide how best to service a client's window of interest, the current Kakadu server implementation now offers a rate-distortion optimization feature. Specifically, the server sequences information which is relevant to the client's window of interest in such a way as to maximize the rate at which the received image quality improves, as measured within the window. To do this, the server requires information collected during compression, which is not defined as part of the JPEG2000 standard. This type of model encourages service providers to ensure that the relevant information is available and to offer the best service possible for the application domain of interest.

In spite of the above arguments, it should be noted that the protocols proposed in this document can actually be used (or abused) to make microscopic requests for specific image elements. Based on the main code-stream header and any relevant tile headers, the client can deduce a small window which is guaranteed to include only one precinct from each resolution level below that specified in the window request. The client can further narrow the request by specifying only one image component and by using the `Cache-Contents:` header to explicitly prevent the server from supplying data for all but one resolution level. The `CUM=yes` query field (see Section 3.1) could even be used to create a string of such microscopic requests, to be served jointly. While this type of construction is allowed by the syntax, it is clearly not encouraged. Only those clients which really know what they are doing should attempt such usage. Moreover, the server is permitted to refuse to accumulate window requests by refusing to process the `CUM=yes` query field.

The third philosophical principle stated above is difficult to escape if servers are to be deployed in resource constrained environments. The amount of memory required to serve a large image region to a client in order of increasing quality, generally grows in proportion to the size of the region. A similar problem arises if clients wish to receive progressively improving image quality from multiple regions simultaneously, if these regions are sparsely distributed throughout the image.

The source of the memory problem is that the server must usually hold the entire image in a file on disk. If this complete representation were ordered in a quality progressive fashion and the image were very large, access to small spatial regions would be particularly inefficient, requiring data to be fetched from numerous locations scattered across the disk. For this reason, large images will most likely be stored using the RPCL progression defined by JPEG2000 part I. In order to construct a quality progressive stream representing a spatial region within the image, the server must first load the entire region into memory, the cost of which grows with the size of the region. Specific implementations might not explicitly load the data into memory, relying instead on disk caching mechanisms, but the cost in memory or disk thrashing cannot be avoided. For this reason, servers must be able to modify the size of an image region and/or the number of image components specified in a window request.

11 Properties of the Kakadu Implementation

11.1 Transport Modes, Proxies and Sessions

Both the Kakadu client (as demonstrated within the "kdu_show" application) and the Kakadu image server, can operate in the following 5 different transport modes.

11.1.1 JPIP-HT Session Mode

The “jpip-ht” protocol variant is described in Section 7. To use this mode, select “jpip-ht” as the session-type in “kdu_show”s URL entry dialog box. The client requests a “jpip-ht” session following the procedure outlined in Section 8.1, specifying `session/jpip-ht` as the acceptable media type. The server creates a session (possibly on a different host) and responds with the session parameters, using the syntax outlined in Section 8.2.

Before the client can issue window requests, it must first establish two persistent TCP connections with the server, which will be attached to the created session. The “primary” TCP channel is used for HTTP request/reply communications, while the “secondary” TCP connection is used for return data/acknowledge communications. It is possible that the primary TCP channel has already been established to retrieve the session parameters, but it is also possible that the session parameters identify a different host or port for window requests within the session.

Each new session is assigned its own server thread, which waits (for a specified maximum amount of time) for the client to complete the two required TCP connections. Both connections are received on the same port (port 80 by default), which simplifies the server and minimizes the risk of problems penetrating firewalls. The first paragraph written by the client on each TCP channel is used to identify the session to which it belongs and whether it is the primary or the secondary TCP channel.

In this mode, the server estimates network conditions and manages the amount of transmitted data which is outstanding (not acknowledged) on the network so as to ensure responsiveness to new client requests, while simultaneously avoiding loss of transport efficiency.

The server terminates the session once either the primary or the secondary channel is closed by the client, or if a session timer expires. The Kakadu server does not currently offer any option to form new TCP connections with the same “jpip-ht” session, once a previously connected channel has been closed. This should cause no problem in practice.

11.1.2 JPIP-H Session (persistent) Mode

This is similar to the “jpip-ht” mode described above, except that all communication proceeds over a single TCP channel, via the HTTP protocol, as described in Section 6. As before, the session is terminated once the TCP channel is closed, either by the client, or by the expiration of a session timer. To use this mode, select “jpip-h” as the session-type in the URL entry dialog box offered by “kdu_show”. Note, however, that the mode may be altered to that described below, if an intermediate HTTP/1.0 proxy is involved in the communication.

In this mode, the client takes responsibility for flow control, supplying a byte limit (`B=<spec>` query field) with each request, so as to avoid too much data being pushed into the network by the server, which would damage responsiveness. The client’s policy is to issue a new request for the same window, once the first third of the response is received from the previous request, unless the user modifies the window of interest, in which case the new request may be issued immediately.

11.1.3 JPIP-H Session (non-persistent) Mode

As noted above, this mode is established automatically if a "jpip-h" session is requested, but the communication is found to involve an intermediate proxy which does not support HTTP/1.1. Perhaps the most significant difference between HTTP/1.0 and HTTP/1.1 is that the former creates a new TCP connection for each request, while the latter maintains a persistent TCP connection by default. Some implementations of HTTP/1.0 support a "keep-alive" request from the client or server, but there are inherent risks with this construction, as noted in RFC2616. For this reason, the Kakadu server and client assume that any HTTP proxy with version less than 1.1 will not support persistent connections.

Once a non-conforming proxy is detected, both the server and the client enter the non-persistent mode, including `Connection: close` header lines with all request and reply paragraphs to ensure that the server, the client and all intermediate proxies follow the same policy of closing the TCP connection after each request/response pair. Even though the connection is not persistent, the session maintained by the server does persist, maintaining its model of the client's cache between requests.

As before, the client manages flow control, using byte limits (`B=<spec>` query fields) to ensure that the server does not push too much data onto the network, damaging responsiveness. Note, however, that in this case the client will not issue a new request until the server's response to a previous request has been received in full. The server actually allows a new TCP channel to be connected before it finishes sending the data from the last request, but the client does not currently attempt to exploit this feature. In practice, non-persistent communication is highly inefficient for networks, which is precisely the reason why HTTP/1.1 establishes persistent connections as the default. As a result, we would expect the non-persistent mode to be used with decreasing frequency as proxy implementations move to HTTP/1.1.

In this mode, the session is closed by the server if a session timer expires, or if the client takes more than a specified amount of time (defaults to 20 seconds) to establish a new TCP connection, after the previous request (and hence TCP connection) was finished. The session is also terminated if an active TCP channel is terminated unexpectedly by the client. Our experience so far has shown this mode to work robustly across networks involving HTTP/1.0 proxies.

It is worth noting that the server may terminate the session if the interactive user becomes idle for too long. Once the server has finished supplying all data relevant to the user's current window of interest, no new client requests will be issued, until the user selects a different window (different spatial region, resolution, or image components). Since there is no open TCP connection between these events, the server will terminate the session if the idle period exceeds its limit (defaults to 20 seconds). This is not an inherently undesirable behaviour, but the Kakadu client will not currently re-establish the session automatically; the user must explicitly open the URL dialog again.

11.1.4 Session-less (persistent) Mode

This mode is offered principally for experimental purposes, since it is not clear what purpose would be served by combining session-less state management with a persistent TCP connection. The mode may be obtained by selecting "none" as the session-type in "kdu_show"s URL entry

dialog. In this case, the client issues window requests directly to the server (or possibly via an HTTP/1.1 proxy), without first requesting any session parameters.

Every request identifies the image, rather than a session ID, and every request must be self-contained, in the sense that `Cache-Contents:` and/or `Cache-Needs:` headers must be used to identify elements which the client already has in its cache, so that they will not be delivered again by the server. As might be expected, this makes the request paragraphs significantly larger than they are when requests are presented within the context of a stateful session.

To implement this mode, the Kakadu server actually creates an internal (hidden) session when it receives the first window request. It then donates the TCP channel to that session, so that all future requests passed over the same persistent TCP connection must refer to the same image. Although the server could maintain state between requests, by virtue of the presence of a persistent TCP connection, it explicitly erases all state information whenever each new request arrives, so as to simulate the behaviour of a state-less server.

As for "jpip-h" sessions, the client manages flow control. However, the client will not send a new request until after it has received the complete response from a previous request. This is important, because the client must identify its (relevant) cache contents in each request. If it sends a new request before waiting for the server's response, the cache contents which it identifies in its new request will not be as complete as they could be, causing the state-less server to send more data than is necessary in response to the new request. The combination of enlarged request paragraphs and no request pipelining, make sessionless operation inherently less efficient than session-based communication.

11.1.5 Session-less (non-persistent) Mode

The only difference between this mode, and that described above is that a new TCP channel is opened for each new request. Once again, each request is fully self-contained, identifying both the image and any existing cache contents which are relevant to the requested window. The Kakadu client and server migrate from persistent session-less communications to non-persistent session-less communications if an intermediate proxy does not offer support for HTTP/1.1. Usually, this happens when one or more old HTTP/1.0 proxies are involved in the communication chain.

This mode is significantly less efficient than any of those described previously, because the server must create a new internal session, opening a handle to the image, for each individual request.

Session-less requests with non-persistent TCP connections make the most sense when only one a single request is to be issued. This non-interactive mode of operation is used by the Kakadu client whenever the original image URL specifies an image window, rather than just the image name. For example, supplying a URL of the form

```
http://host/big.jp2?R=640,480&O=100,100&S=200,150
```

or

```
http://host/myscript?Image=big.jp2&R=640,480&O=100,100&S=200,150
```

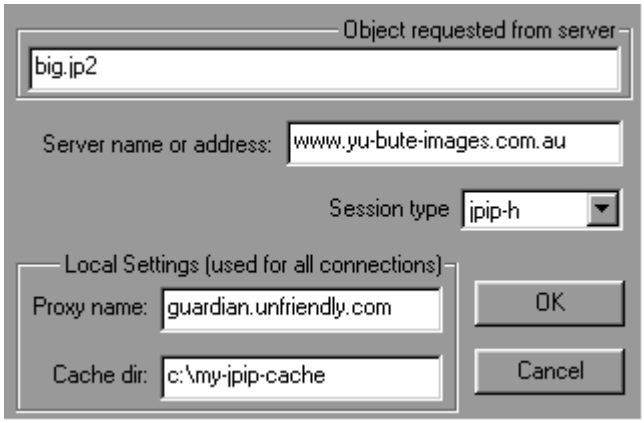
on the "kdu_show" command line will cause a single session-less request for the same URL to be issued to the server; the Kakadu client will refuse to accept or deliver any further window requests in this case.

11.2 URL Entry with "kdu_show"

The purpose of this section, is to illustrate some of the various ways in which URL's may be supplied to "kdu_show" to establish communication with a remote server. We do this through three examples.

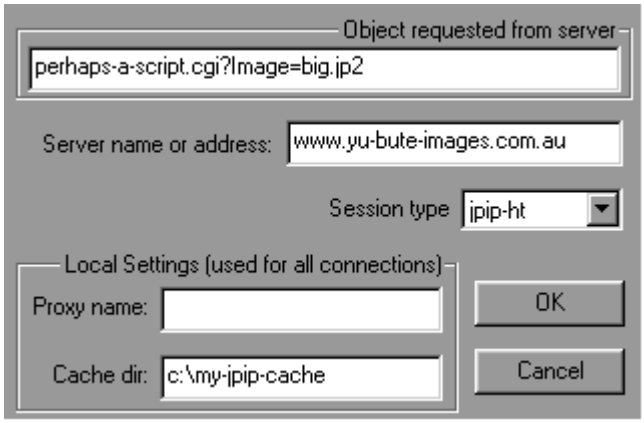
- The figure below depicts the contents of the URL entry dialog offered by "kdu_show". In the example, a "jpip-h" session is being requested for interacting with an image named "big.jp2". The session request is to be delivered to the host "www.yu-bute-images.com.au". It is possible that this machine will not be used to serve the actual image requests, in which case the session parameters returned to the client will specify a different machine, but the user need not be aware of this detail. All HTTP communications are to be forwarded via the HTTP proxy at "guardian.unfriendly.com".

  The image data received during the session is to be stored in an appropriate file, within the directory "c:\my-jpip-cache", unless the server could not supply a unique *Image-ID* for verifying that future browsing sessions refer to the same image. Likewise, any image data cached in this directory which matches the *Image-ID* supplied by the server during session establishment will be used in rendering the image and avoiding the transmission of redundant information by the server.



- In the example below, a "jpip-ht" session is requested, again from the server at "www.yu-bute-images.com.au". The image name in this case is identified using an Image=<spec> query field in the resource name. The Kakadu server pays no attention to the rest of the resource name, but the request might be forwarded to the Kakadu server from a CGI script, processed say by a standard Apache web server. In this case, the server's response, containing the session establishment parameters, would be passed back through the Apache web server and the client would use them to establish the ongoing image communication session directly with the JPIP image server (e.g. an instance of "kdu_server" running on some machine).

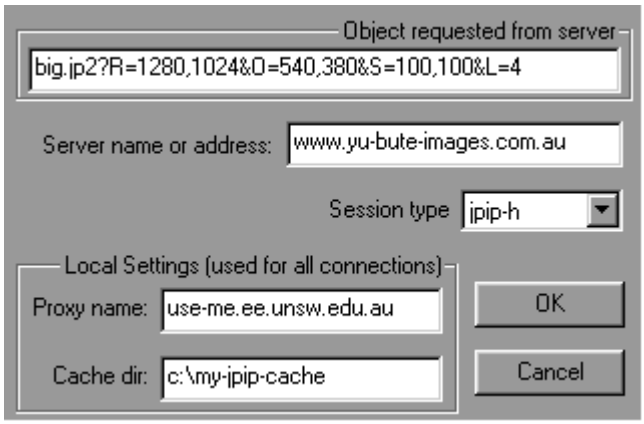
  In this example, no proxy is supplied, although the session request could be delivered over an HTTP proxy if desired. In any event, once the session parameters are received, the Kakadu client will form persistent TCP connections directly with the image server for all ongoing communication.



- In this last example, shown below, the requested object includes a query string which identifies a window into the full image. In this case, the query string specifies that the image should be accessed at a resolution which would fill a 1280 wide by 1024 high display, were it to be rendered in full, retrieving only the compressed data which is required to reconstruct a 100x100 region at the centre of the image, up to and including the 4th quality layer.

  When the Kakadu client processes a URL of this form, it makes a single session-less request to the server, using HTTP for all communication, and closing the TCP connection once the server's response is complete. New window requests from an interactive user will not be posted to the server. The session-type identified in the dialog box is ignored, since there is no point in establishing a session for a once-off request, although the client could establish a session if it wanted to.

  All received data are recorded in an appropriate file within the supplied cache directory, for future browsing. Moreover, if the cache already contains information for this image, it will be used to improve the image rendering, regardless of the window specified in the actual request sent to the server. Note, however, that the client's cache contents are not supplied with the client's request, so the server will issue a full response to the request, unless the connection is terminated prematurely by the client. This makes the server's response cacheable by intermediate HTTP proxies.



Before concluding this section, it is worth noting that the "kdu_show" application can also be supplied with an image URL on the command line. If the first token found on the command line has the form

```
jpip://<server name>/<object>
```

or

```
http://<server name>/<object>
```

an attempt will be made to process it as a JPIP image request, rather than a local image file. In this case, the `<object>` portion of the URL is treated in the same way as the first line in the URL entry dialog. The session-type and any proxy or cache directories for such requests are obtained from the most recent values supplied via the URL entry dialog, since these are cached in the system registry.

11.3 Preparing Images for Serving

The Kakadu image server can deliver pretty much any JPEG2000 Part 1 code-stream, or JP2 file to a remote client, regardless of the code-block or precinct dimensions, number of quality layers, image bit-depth, number of image components, or whether the image was tiled or untiled. Since the input image might be untiled, having precincts whose dimensions are those of the entire image, the Kakadu server generally transcodes the input image into one having much smaller precincts, which can then be delivered incrementally in accordance with the remote client's spatial region of interest.

The transcoding is performed on the fly, and has relatively little impact on computation or memory resources. This is because we do not fully decode and re-encode the code-block bit-streams; instead, the existing code-block contributions are simply re-packaged into the smallest precincts which are consistent with their dimensions. Precincts were introduced into the JPEG2000 standard with this kind of transcoding in mind.

Even though the Kakadu server can handle virtually any input image, not all input compressed images are equally suitable for interactive browsing. The most appropriate images for use with the Kakadu client-server tools are those having the following code-stream attributes.

Multiple quality layers: This allows quality progressive transmission of the image elements which are relevant to any given client window and prevents individual compressed data packets from becoming too large. Since these properties may be desirable over a wide range of resolutions, a large number of quality layers is generally desirable. Generally speaking, successive layers should have bit-rates which are separated roughly logarithmically, with 2 to 5 layers per octave change in bit-rate.

The smallest layer bit-rate should be no larger than $2^{-(2+r)}$ bits/pixel, where *r* is the number of different resolutions at which full quality scalable browsing is desired. This may be accomplished by supplying the following arguments to "kdu_compress," where we are assuming that full scalability is required at resolutions down to one 1/32 of the full image width and height.

```
Creversible=yes Clevels=7 Clayers=30 -rate -,0.008
```

Alternatively, the distortion-length slopes of the individual quality layers may be controlled directly during compression. Kakadu's compression utilities work with a logarithmic representation of the distortion-length slope, given by $2^{16} + 256 \log_2 (\Delta\text{MSE} / \Delta\text{Length})$. In this context, you would do best to log slope values which are separated by between 100 and 300. This may be accomplished by supplying the following arguments to "kdu_compress."

```
Creversible=yes –slope 0,48500,48700,48900,49100,...,55000
```

Multiple precincts: If each resolution is represented by only one precinct within the code-stream, the server will need to read the entire precinct in order to extract the relevant code-blocks and transcode them into smaller precincts for serving. For large images, the amount of server memory consumed by this task could easily become prohibitive. To avoid this difficulty, it is recommended that precinct dimensions should be selected in the range 128x128 to 256x256, with larger precincts being used only for the highest resolution levels of very large image (e.g., images whose original size runs into the Gbytes). As an example, this may be accomplished by supplying the following arguments to "kdu_compress."

```
Clevels=7 Cprecincts={256,256},{256,256},{128,128}
```

No tiling or large tiles: Small tiles do not appear to have significant benefits for efficient remote browsing of JPEG2000 images. Instead, they significantly add to the amount of work the server must do to serve client requests at low image resolutions; these inevitably involve a large number of tiles. To avoid these difficulties, it is recommended that you leave the image untiled or, if tiles are required for other reasons, select tiles of size 1024x1024, the largest size compatible with a tiled Profile-1 code-stream.

Code-Blocks: Code-blocks of size 64x64 or 32x32 are both suitable for remote image browsing, but the smaller dimensions are generally preferable to improve the efficiency with which small image windows are serviced. The following argument to "kdu_compress" is appropriate:

```
Cblk={32,32}
```

Pointer Information: Be sure to include PLT (Packet-Length Tile-Part) marker segments in the code-stream so that the server can randomly access source image packets as-needed. Otherwise, the server will need to sequentially parse and buffer all of the precincts in all relevant tiles (remember that the image will often be untiled). To get "kdu_compress" to include PLT marker segments, use the following argument.

```
ORGgen_plt=yes
```

Progression Order: To enable the server to efficiently access precincts of interest from the source code-stream, all packets from those precincts should appear contiguously within the file. In fact, the Kakadu tools will only use PLT marker segments to randomly access code-stream elements if all packets of each precinct appear contiguously within the code-stream; otherwise, multiple pointers must be remembered for each precinct. The three progression orders which are consistent with this requirement are those designated RPCL, CPLR and PCRL, of which RPCL is probably to be preferred. This may be accomplished by supplying the following argument to "kdu_compress."

```
Corder=RPCL
```

Tile-Parts: Even though the image will frequently be untiled (i.e., consisting of a single tile), tile-parts still represent a useful device for distributing PLT marker segments information throughout the code-stream, thereby reducing the amount of pointer information which the server must digest when an image is first opened for browsing. In combination with the RPCL progression order mentioned above, I have found it useful to insert tile-parts at each resolution boundary, although other schemes may be warranted for truly enormous images. The following commands may be supplied to "kdu_compress."

```
ORGtparts=R
```

11.4 Disk Caching and Unique Image Identifiers

The Kakadu server generates a unique identifier for each image, which is used by the client to determine whether or not it can re-use data which it has cached to disk after a previous interactive browsing session. The exact format of the Image-ID string is not defined by the protocol, but the Kakadu server uses a 32 character hexadecimal representation of a 128-bit unsigned integer. The integer is formed from the file's full path name on the server, the file's last modification time and the server's MAC address, using the AES (Rijndael) encryption algorithm to confound this information in the interest of privacy.

In many applications, it may be desirable to serve exactly the same image from multiple machines, where the Image-ID string should be independent of the machine which is performing the service. Since different machines will generally have different MAC addresses and may store the image in different locations, the mechanism described above will generate different Image-ID strings on each platform. To avoid this difficulty, the Kakadu server writes a simple text file containing the Image-ID string into the same directory as the image itself. This file has the same name as the image, except that the original image suffix is prefixed by an underscore character instead of the period, and a new suffix of ".kid" (for "Kakadu ID") is appended. If the server finds such a file, it takes the Image-ID from the file, rather than generating it from scratch. Thus, to ensure that multiple platforms report the same Image-ID string, it is sufficient to copy the ".kid" file along with the image file itself, to each machine which may provide the service.

The Kakadu client, embodied by the self-contained "kdu_client2" object, and demonstrated by the "kdu_show" application, will store the compressed data recovered while browsing an image to disk, provided the following conditions are satisfied: 1) a valid cache directory must be supplied; and 2) the server must supply an Image-ID string. In "kdu_show", the cache directory is configured via the URL entry dialog. Clearing the relevant field in the URL dialog will prevent image data from being saved to disk and will also prevent previously cached image data from being retrieved from disk.

In session-oriented communications (i.e., "jpip-h" or "jpip-ht" sessions), the Image-ID string is supplied by the server during session establishment. When no session is used (see Sections 11.1.4 and 11.1.5), the client requests information about the Image-ID by including the field, `IID=0`, in the query string of its first request. The server's reply should contain an `Image-ID:` header if an identifier is available; the Kakadu server always makes one available.

Cache files written by the Kakadu client have names formed by appending the suffix, ".kic" (for "Kakadu Image Cache") to the Image-ID string itself. If a cache directory is available, and a cache file is found which matches the Image-ID string supplied by the server, the cache file is read and its contents are used for the following two purposes: 1) to immediately enhance the image being rendered locally by the user's application (e.g., "kdu_show"); and 2) to generate `Cache-Contents:` headers which will avoid the transmission of redundant information by the server.

The contents of an existing cache file are used even in non-interactive communications, where the image URL is supplied complete with window coordinates, as demonstrated in the last example of Section 11.2, but cache contents are not signalled to the server in this case.

11.5 Server Delegation and Load Sharing

The Kakadu server offers a useful feature for dynamically delegating session requests (either "jpip-h" or "jpip-ht" sessions) to other servers. In a typical application, a front-end server may be configured on a publically known host to listen on port 80 for session requests. When a session request is received, the server passes the request on to another host, not usually with a fixed or public IP address, waiting to receive an HTTP reply containing the session parameters, as described in Section 8.2. If successful, the session parameters are passed on to the original client and the TCP connection is closed. The client then forms its own connection with the image server identified via the mandatory `Session-Host:` header in the session parameter list.

The Kakadu server provides multiple threads for handling the initial processing of new TCP connections, allowing it to process and attempt to delegate multiple connection requests concurrently. In this way, a single front-end server should be able to handle thousands of incoming requests per second, delegating the real work to alternate machines. The delegation algorithm employs a load sharing algorithm, which re-uses delegation hosts which are known to be responding in a round-robin fashion, based on load sharing factors supplied when the server is launched. By way of example, the following command line causes 15 session requests to be delegated to host "private1" for every 5 requests which are delegated to host "private2", in both cases contacting the delegation server on port 80.

```
kdu_server –delegate private1:80*15 –delegate private2:80*5
```

If a delegation server is found not to respond, the next host is tried in order. Hosts which have failed to respond are occasionally retried, based on the load sharing factors. In the above example, if the host "private2" failed to respond, it would not be retried after the next 15 requests had been delegated to "private1."

11.6 Support for MetaData

Neither the "kdu_server" nor the "kdu_show" application currently makes use of meta data-bins or catalog data-bins, except for meta data-bin 0, which is invariably used to convey the mandatory elements of a JP2 file's preamble (see Section 5.2.3). Nevertheless, the core objects on which these applications are built do provide substantial facilities for sequencing, transporting and caching meta data-bins and catalog data-bins.

The "kdu_serve2" object, on which the "kdu_server" application relies for sequencing incremental image contributions in an appropriate manner and for packing these contributions into data chunks, provides a facility for registering ancillary data-bins for transport. Specifically, the "kdu_serve2::push_ancillary_data" function allows the application to supply meta data-bins and catalog data-bins, together with prioritization and scoping information. Each of these ancillary data-bins may be assigned a spatial scope and a range of image components, within which the data-bin should be considered relevant. The data-bin becomes a candidate for inclusion in server messages sent to the client only if its scope intersects the window requested by the client.

In addition to scope, each ancillary data-bin (meta data-bin or catalog data-bin) may be assigned a sequencing priority, S. An ancillary data-bin with sequence number S will not be considered for delivery to the client until the first S packets of all precinct data-bins which are relevant to the client's requested window have been sequenced into the message stream. For

reference, code-stream header data-bins have an implied sequencing number of 0 and meta data-bin 0 has a sequence number of –1.

Delivery of ancillary data-bins and catalog data-bins to the client is controlled exclusively through these scope and sequencing conventions. The only way a client can prevent relevant ancillary data-bins from being delivered in response to a request is by listing the elements in `Cache-Contents:` headers, such as

```
Cache-Contents: M*,C*
```

On the other hand, the server application may choose supply ancillary data-bins to the "kdu_serve2" object only on demand, once it can be sure that the client is interested in them. The "kdu_serve2::push_ancillary_data" function may be called at any time. In fact, an ancillary data-bin may be left open and incrementally augmented whenever the server application sees fit. At the appropriate point, the server will deliver all available data for such data-bins to the client, but will not mark the data-bin as completed.

The "kdu_client2" object processes received ancillary data-bins, adding them to the evolving cache structure maintained by the "kdu_cache2" object. The information in these data-bins may be retrieved by an application on demand, and it will automatically be saved to the disk cache and subsequently retrieved, if caching is enabled by the availability of a cache directory and a suitable Image-ID string (see Section 11.4).

11.7 Cross-Platform Support

The Kakadu server and client implementations both employ elements which are specific to the WIN32 operating system. However, both the client and server implementations are built on top of platform independent base objects, which provide core services. In particular, the "kdu_server" application is built on top of the "kdu_serve2" base object, whose implementation contains no WIN32 specifics. Similarly, the "kdu_client2" object, which implements a complete self-contained client, is built on top of the platform independent "kdu_cache2" object.

With few exceptions, the platform dependent elements in "kdu_server" and "kdu_client2" have been carefully isolated into objects with well-defined interfaces, so that the amount of code which must be ported to different platforms can be very small. The two key platform dependent aspects of the code are multi-threading and networking. With very few exceptions, all network interaction proceeds via instances of the class, "kd_tcp_channel," whose members would need to be ported to other platforms of interest. Multi-threading elements are wrapped up in functions with names like "acquire_lock", "release_lock", "signal_event", "wait_event" and "thread_start", which can similarly be ported to other platforms which offer comparable services.

11.8 Rate-Distortion Optimized Delivery

To conclude our brief account of the Kakadu JPIP implementation, we make mention of the rate-distortion optimization feature now offered by the server. Previous Kakadu server implementations collected the set of all compressed data precincts which are relevant to the client's window of interest, sequencing them layer by layer into server messages to be delivered to the client.

Of course, only relevant precinct data is sent, but this can and often does include precincts whose code-blocks contribute to the reconstruction of image regions which do not belong to the client's window of interest. In earlier implementations, all relevant precincts were assigned equal importance and their compressed data was scheduled in order, from the lowest resolution levels to the highest resolution levels, within each quality layer, moving through the layers one by one. This mode of behaviour may still be accessed by specifying the `-ignore_relevance` command line option to "kdu_server."

By default, the Kakadu server now does two things to sequence more relevant data before less relevant data. Firstly, within each quality layer, the precincts are considered in order of their relevance, rather than in order of their resolution. The relevance, r_i, of a precinct, i, is calculated as

$$r_i = \frac{\left\| A_i \cap W_{R_i} \right\|}{\left\| A_i \right\|}$$

where A_i is the spatial region occupied by precinct i on its its tile-component-resolution, R_i, and W_R is the projection of the samples in the subbands belonging to resolution *R*, which contribute to the reconstruction of the client's spatial window of interest, onto the coordinate system of resolution *R*. The intersection, $A_i \cap W_{R_i}$, is guaranteed to be non-empty for any precinct which is relevant to the window, *W*. Clearly, then, the relevance factor satisfies $0 < r_i \leq 1$.

Interestingly, lower resolution precincts tend to be only partially relevant to small windows, defined at high resolution, so that this scheduling policy actually tends to sequence the lower resolution precincts later than the most relevant higher resolution precincts.

The second and much more significant feature of the Kakadu server's relevance sensitive sequencing algorithm is available only if the source code-stream contains auxiliary information identifying the distortion-length slope thresholds which were used by the compressor when generating the code-stream layers. This information is now automatically recorded by Kakadu compression utilities, in a special COM (comment) marker segment in the code-stream's main header. You may inspect the contents of these COM marker segments by selecting the *File®Properties* menu item, from within "kdu_show". When this information is available, the Kakadu server is able to modify the sequencing of information from different layers within different precincts, so as to maximize the rate at which the expected image distortion is reduced within the client's window of interest, as a function of the amount of data transmitted.

The basis of this rate-distortion optimization feature is the reasonable assumption that the impact of a single packet from precinct *i* on the MSE distortion within the client's window of interest, is equal to r_i times the impact which this same packet has on the MSE distortion over the entire image. Now the code-block contributions which were included in code-stream layer l have distortion-length slopes (reduction in MSE, divided by increase in code length) no smaller than s_l, where s_l is the slope threshold recorded by the compressor in the COM marker segment mentioned above. Modifying the code-block distortion-length slopes to account for relevance to the client's current window requires only that they be multiplied by r_i.

Of course, we do not have the actual code-block distortion-length slopes, but if the slopes were multiplied by r_i, the corresponding packet data contributions would have been included in the quality layer having a slope threshold of $s_l r_i$, rather than s_l, assuming that such a layer exists. Following this reasoning, the server's R-D optimal sequencing algorithm seqences the packet data in layer l of precinct *i*, as though it had appeared instead within layer l', where $l' \geq l$ is the layer whose slope threshold satisfies $s_{l'+1} < s_l r_i \leq s_{l'}$. Note that slope thresholds, s_l, invariably decrease as l increases. The Kakadu server employs a number of implementation tricks to ensure that the appropriate sequence can be determined with very little computational effort. Of course, this strategy is most effective when the source code-stream contains multiple layers, whose cumulative bit-rates are spaced reasonably closely.

Preliminary investigations suggest that the R-D optimal sequencing strategy can increase the PSNR (reduce the MSE) by up to 8dB within the client's window of interest, at any given point in the transmission sequence. To investigate the performance of this feature yourself, you may like to perform an experiment similar to the following. Using an image compressed with at least 2 (preferably 4) quality layers per octave change in bit-rate, containing the informative COM marker segment now produced by the "kdu_compress" application, start the "kdu_server" application and supply "kdu_show" with a URL of the following form

```
jpip://my_host/bike_image.jp2?R=2048,2560&O=1024,1024&S=230,230&B=20000
```

This causes a single request to be issued, returning at most 20000 bytes of compressed data, relevant to the supplied window. Try sending requests with different byte limits and, in each case, saving the result from "kdu_show" and measuring the MSE associated with the requested image region. Now repeat the experiment supplying the `-ignore_relevance` command line option to the "kdu_server" application. Comparing the MSE's obtained with and without relevance sensitive sequencing should provide an indication of the usefulness of this feature.