CCR Exchange Protocol (CXP)

CXP is a two party peer to peer protocol that moves CCR-family data and other XML based structures across the Internet between co-operating Sending and Receiving systems from multiple vendors. CCR-related data includes PDF, DICOM, and other documents referenced from within a CCR, and a CCR itself.

A Transfer is the movement of a CCR and Referenced documents or other XML based data. In CXP, we refer to whichever party is holding and moving the CCR as the Sender, and the party that is accepting and potentially storing the CCR as the Receiver.

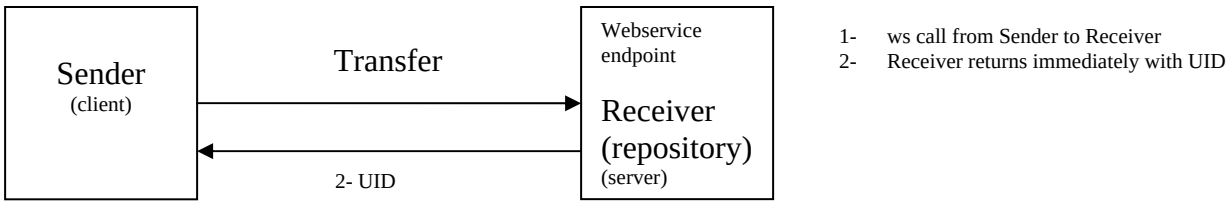

Figure 1

A Transfer is initiated by a Transfer Command calling a pre-registered webservice endpoint URL. As shown in figure 1, a basic Transfer flows from Sender to Receiver, is safely stored by the Receiver, and a document key known as the UID flows back from the Receiver to the Sender. The UID is defined by the CXP protocol and must be precisely implemented by both Sender and Receiver[1].

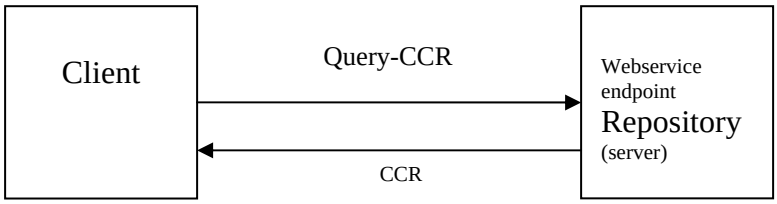

Figure 2

As shown in figure 2, a specific CCR can be located via the Query-CCR Command by passing in the UID. To retrieve attachments, a similar Query-XML Command is utilized.

Indexing and Retrievals

CCRs will be stored and indexed in different ways by different Vendors, but each CCR is accessible by UID, which is the SHA-1 hash of the CCR itself. There are no other prescribed or mandatory keys, tags, or any identifying characteristics other than data present in the CCR itself and some vendors may accept CCRs alone, with no additional metadata. Vendors can index whatever elements of the CCR they choose and may accept

[1] See appendix for open source javascript

an optional XML Data Block[2] to support the client authorization and query requirements when those elements are not accessible in the CCR.

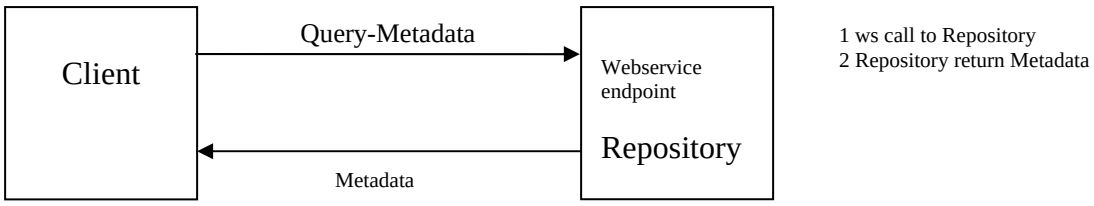

Figure 2

A Query-MetaData Command can be issued to retrieve metadata about CCRs. Different Vendors will support different Query options depending upon, security, and flexibility preferences.

Apart from the UID, which can be computed by either the client or the repository, the identifiers used in the queries are determined by the Receiver. The scope of the identifier is determined by the repository – some identifiers may be permanent; others are time limited; still others may be one-time access keys. Defining these further is outside the scope of this proposal.

Interaction with Higher Level Protocols

There will be higher level protocols constructed on top of CXP. For example, there may be protocols for registration of Patients and other administrative tasks as well as for moving CCRs and attachments. In all cases these are layered on top of CXP as follows:

- CCRs and an optional XML-Data block are passed back and forth in standard form
- Administrative commands encode their functions and return status in an XML-Data block
- CCR Referenced documents may be transferred along with the CCR as XML-Data or separately by non-CXP methods.
- CCR References moving with the CCR are base-64 encoded and passed as part of XML-Data. This is the default. CCR References not moving with the CCR may need additional standards and protocols.

[2] We may want to enter a constraint here that the XML block should not replicate data that is in the CCR. This is attractive because it means that there isn't the possibility of inconsistencies between the CCR and the XML block. It's unattractive because for some vendors the XML block may be condensed version of the CCR that parses more quickly or it may be some other standard XML format which requires (say) patient demographics. This should be nailed down by discussions with the accelerator group.

- SSL or TLS may be utilized and is encouraged between parties, but it is not required.
- Encrypted CCRs are permitted in CXP.

Thus the XML-Data block is where all non-CCR data that needs to be passed between systems is expressed. This is application specific.

Security

A CXP recipient vendor shall accept CCRs encrypted per the ASTM CCR spec. Status responses indicate if insufficient information was provided in order to issue a confirmation to the sender. In some cases, transfer of encrypted CCRs may require pre-registration of the patient or other Actors. These pre-registration steps are considered higher level protocols beyond CXP.

The CXP sender must accept responsibility for release of information to the recipient. Patient consent to information release is the responsibility of the sender.[3]. Consent and the key management are outside of CXP. Another example might be implied consent between two CXP systems that have a HIPAA provider or business associate relationship and that authenticate each other via TLS. The HIPAA relationship is beyond CXP. Other consent mechanisms may exist, all outside CXP.

A CXP receiver is expected to manage its own security independent of the sender system or vendor. There is no defined responsibility for a recipient to publish general policies or to handle security for a particular transfer in a particular way. If the sender needs specific security information to manage its own policy, then the negotiation of these assertions is beyond the scope of CXP.

Apart from the use of SSL and TLS there is no authentication of the machine at the other end of the CXP connection. A Sender and Receiver can agree to use extra fields in the XML Data Block to pass additional authentication data such as a shared secret in order to provide additional validation to the transaction.

Implementation

CXP is based on SOAP over HTTP(S) or alternatively POST over HTTP(S). A Sender or Receiver may choose to implement either or both alternatives. This is a static choice made by the Vendor during installation. If implementing both, a separate set of URL endpoints will be utilized.

[3] For example, use of a recipient's public key as derived from a consent form explicitly signed by the patient is one way to satisfy this legal mandate

For a Sender to reliably connect and transfer a CCR to a Receiver, the Receiver must have a fixed public IP address and must publish the URL(s) (via email, word of mouth, etc) prior to accepting CXP connections.

Those operations such as Queries that do not imply a Sender and Receiver, but rather a Client and a Repository, the Repository is required to have a fixed IP address.

Commands

All commands map directly to SOAP or POST over HTTP. For compatibility, only a single output argument is utilized:

The general form is:

Xml-Return-Data = CXP-Command([CCR],[XML-Data]);

- ALL invocations of cxp-command return an XML data structure. The formats of this data structure are discussed below, and in the Errors section.
- Either the XML-Data block, OR the CCR must be present. Otherwise an encoded 404 error is returned to the caller.
- If the XML-Data block is absent, and operation-code of 'Transfer' is implicitly assumed. Otherwise the operation-code in the XML-Data block is utilized and other data values are defaulted as described elsewhere in this spec.
- Some operation codes return a real CCR as the XML-Return-Data, others will generate other XML return structures. This will evolve other time.

The same form of command is used in both directions. When a Notification of an incoming CCR is delivered via SOAP or POST, the argument list is exactly as described above.

Operation-code	**CCR**	**XML-Data Block**	**XML-Returned-Data-Block**
Transfer	Yes	Optional, if present includes attachments	Returns UID or error
Query	No[4]	Yes, includes	Returns success or failure for query

[4] If a Query returns a CCR plus attachments, then the simple form of Query can not be used to get the attachments. Instead, a notification endpoint is specified in the XML Data Block supplied with the Query. The endpoint will be called with the CCR and attachments encoded as a CCR and an XML Data Block that can be decomposed into separate attachments

		query params	along with either CCR or XML data

Error Handling

Any invocation of CXP-command can return all O/S level TCP and HTTP errors. Beyond that, everything is returned as valid XML. All errors come back as valid XML structures.

XML-DATA

The XML-Data block passed between Sender and Receiver has different fields of interest depending on the specific operation request. The general structure, which is always unencrypted, is

```
<cxp>
        <opcode>opcode</opcode>
        <Files>
                <File>
                        <FileName>as-named-in-ccr</FileName>
                        <FileType>one of: pdf, dicom, etc </FileType>
                        <FileContents>base64-encoded-bits</FileContents>
                </File> …….
        </Files>
        <QueryNotificationURL>
                https://mumbler.mumbler.mubler/query.xyz?data=foo
        </QueryNotificationURL>
        <QueryString>
                either a UID , and possibly vendor dependent variations
        </QueryString>
</cxp>
```

The only mandatory field is operation-code. The other fields are present or not, depending on the needs of the operation-code. The xml data block is naturally extensible by adding additional tags which must be ignored if not understood by the Receiver.

Trademarks and Intellectual Property

CXP (or whatever the CCR Accelerator Group chooses to call this) is an alternative to XML File Import or Export from a Vendor System when the file is a CCR. CXP provides the alternative of a Web Service endpoint for the Import or Export target and, beyond that, offers the minimum required set of tracking and status protocols to enable reliable transfer of responsibility between two cooperating vendors. It will be useful for either ASTM or AAFP or the CCR Accelerator Group to take ownership and control of the trademark for Web transport via CXP in order to improve the rate of adoption in the marketplace.

Appendix: SHA-1 Implementation in Javascript

We have included the SHA-1 code that can be used by CXP Javascript Clients:

```
/*
 * A JavaScript implementation of the Secure Hash Algorithm, SHA-1, as defined
 * in FIPS PUB 180-1
 * Version 2.1 Copyright Paul Johnston 2000 - 2002.
 * Other contributors: Greg Holt, Andrew Kepert, Ydnar, Lostinet
 * Distributed under the BSD License
 * See http://pajhome.org.uk/crypt/md5 for details.
 */

/*
 * Configurable variables. You may need to tweak these to be compatible with
 * the server-side, but the defaults work in most cases.
 */
var hexcase = 0;  /* hex output format. 0 - lowercase; 1 - uppercase        */
var b64pad  = ""; /* base-64 pad character. "=" for strict RFC compliance   */
var chrsz   = 8;  /* bits per input character. 8 - ASCII; 16 - Unicode      */

/*
 * These are the functions you'll usually want to call
 * They take string arguments and return either hex or base-64 encoded strings
 */
function hex_sha1(s){return binb2hex(core_sha1(str2binb(s),s.length * chrsz));}
function b64_sha1(s){return binb2b64(core_sha1(str2binb(s),s.length * chrsz));}
function str_sha1(s){return binb2str(core_sha1(str2binb(s),s.length * chrsz));}
function hex_hmac_sha1(key, data){ return binb2hex(core_hmac_sha1(key, data));}
function b64_hmac_sha1(key, data){ return binb2b64(core_hmac_sha1(key, data));}
function str_hmac_sha1(key, data){ return binb2str(core_hmac_sha1(key, data));}

/*
 * Perform a simple self-test to see if the VM is working
 */
function sha1_vm_test()
{
  return hex_sha1("abc") == "a9993e364706816aba3e25717850c26c9cd0d89d";
}

/*
 * Calculate the SHA-1 of an array of big-endian words, and a bit length
 */
function core_sha1(x, len)
{
  /* append padding */
  x[len >> 5] |= 0x80 << (24 - len % 32);
  x[((len + 64 >> 9) << 4) + 15] = len;

  var w = Array(80);
  var a =  1732584193;
  var b = -271733879;
  var c = -1732584194;
  var d =  271733878;
  var e = -1009589776;

  for(var i = 0; i < x.length; i += 16)
  {
    var olda = a;
    var oldb = b;
    var oldc = c;
    var oldd = d;
    var olde = e;

    for(var j = 0; j < 80; j++)
    {
      if(j < 16) w[j] = x[i + j];
      else w[j] = rol(w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16], 1);
      var t = safe_add(safe_add(rol(a, 5), sha1_ft(j, b, c, d)),
```

```
                        safe_add(safe_add(e, w[j]), sha1_kt(j)));
      e = d;
      d = c;
      c = rol(b, 30);
      b = a;
      a = t;
    }

    a = safe_add(a, olda);
    b = safe_add(b, oldb);
    c = safe_add(c, oldc);
    d = safe_add(d, oldd);
    e = safe_add(e, olde);
  }
  return Array(a, b, c, d, e);

}

/*
 * Perform the appropriate triplet combination function for the current
 * iteration
 */
function sha1_ft(t, b, c, d)
{
  if(t < 20) return (b & c) | ((~b) & d);
  if(t < 40) return b ^ c ^ d;
  if(t < 60) return (b & c) | (b & d) | (c & d);
  return b ^ c ^ d;
}

/*
 * Determine the appropriate additive constant for the current iteration
 */
function sha1_kt(t)
{
  return (t < 20) ?  1518500249 : (t < 40) ?  1859775393 :
         (t < 60) ? -1894007588 : -899497514;
}

/*
 * Calculate the HMAC-SHA1 of a key and some data
 */
function core_hmac_sha1(key, data)
{
  var bkey = str2binb(key);
  if(bkey.length > 16) bkey = core_sha1(bkey, key.length * chrsz);

  var ipad = Array(16), opad = Array(16);
  for(var i = 0; i < 16; i++)
  {
    ipad[i] = bkey[i] ^ 0x36363636;
    opad[i] = bkey[i] ^ 0x5C5C5C5C;
  }

  var hash = core_sha1(ipad.concat(str2binb(data)), 512 + data.length * chrsz);
  return core_sha1(opad.concat(hash), 512 + 160);
}

/*
 * Add integers, wrapping at 2^32. This uses 16-bit operations internally
 * to work around bugs in some JS interpreters.
 */
function safe_add(x, y)
{
  var lsw = (x & 0xFFFF) + (y & 0xFFFF);
  var msw = (x >> 16) + (y >> 16) + (lsw >> 16);
  return (msw << 16) | (lsw & 0xFFFF);
}

/*
 * Bitwise rotate a 32-bit number to the left.
```

```
 */
function rol(num, cnt)
{
  return (num << cnt) | (num >>> (32 - cnt));
}

/*
 * Convert an 8-bit or 16-bit string to an array of big-endian words
 * In 8-bit function, characters >255 have their hi-byte silently ignored.
 */
function str2binb(str)
{
  var bin = Array();
  var mask = (1 << chrsz) - 1;
  for(var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |= (str.charCodeAt(i / chrsz) & mask) << (24 - i%32);
  return bin;
}

/*
 * Convert an array of big-endian words to a string
 */
function binb2str(bin)
{
  var str = "";
  var mask = (1 << chrsz) - 1;
  for(var i = 0; i < bin.length * 32; i += chrsz)
    str += String.fromCharCode((bin[i>>5] >>> (24 - i%32)) & mask);
  return str;
}

/*
 * Convert an array of big-endian words to a hex string.
 */
function binb2hex(binarray)
{
  var hex_tab = hexcase ? "0123456789ABCDEF" : "0123456789abcdef";
  var str = "";
  for(var i = 0; i < binarray.length * 4; i++)
  {
    str += hex_tab.charAt((binarray[i>>2] >> ((3 - i%4)*8+4)) & 0xF) +
           hex_tab.charAt((binarray[i>>2] >> ((3 - i%4)*8  )) & 0xF);
  }
  return str;
}

/*
 * Convert an array of big-endian words to a base-64 string
 */
function binb2b64(binarray)
{
  var tab = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  var str = "";
  for(var i = 0; i < binarray.length * 4; i += 3)
  {
    var triplet = (((binarray[i   >> 2] >> 8 * (3 -  i   %4)) & 0xFF) << 16)
                | (((binarray[i+1 >> 2] >> 8 * (3 - (i+1)%4)) & 0xFF) << 8 )
                |  ((binarray[i+2 >> 2] >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32) str += b64pad;
      else str += tab.charAt((triplet >> 6*(3-j)) & 0x3F);
    }
  }
  return str;
}
```