This is the design sketch for v2 of the Stats interface.

The present interface

From http://dev.w3.org/2011/webrtc/editor/archives/20121115/webrtc.html

```
callback RTCStatsCallback = void (RTCStatsElement[] statsElements,
MediaStreamTrack? selector);

dictionary RTCStatsElement {
    RTCStatsReport local;
    RTCStatsReport remote;
};

interface RTCStatsReport {
    readonly attribute long timestamp;
    any getValue (DOMString statName);
};
```

Example code:

```
var baseline, now;
var selector = pc.remoteStreams[0].audioTracks[0];

pc.getStats(selector, function (stats) {
    baseline = stats;
});

// ... wait a bit
setTimeout(function () {
    pc.getStats(selector, function (stats) {
        now = stats;
        processStats();
    });
}, aBit);

function processStats() {
    // Real code would:
    // - Check that timestamp of “local stats” and “remote stats”
    //   are reasonably consistent.
    // - Sum up over all the elements rather than just accessing
    //   element zero.
```

```
        var packetsSent = now[0].remote.getValue("packetsSent") -
                baseline[0].remote.getValue("packetsSent");

        var packetsReceived = now[0].local.getValue("packetsReceived") -
                baseline[0].local.getValue("packetsReceived");

        // if fractionLost is > 0.3, we have probably found the culprit
        var fractionLost = (packetsSent - packetsReceived) / packetsSent;
    }
```

There’s also draft-alvestrand-rtcweb-stats-registry-00, which suggests some specific stats.

The feedback from Lyon

- The local/remote distinction is not crucial enough to keep. Separate objects (tied by some ID) is better.
- There needs to be a naming structure that allows things to refer to each other.
- Some attributes are naturally multivalued. They should be allowed to be multivalued.
- Returning the selector has no benefit. If the user wants it, he can pass it to the callback using a closure.
- A fully worked complex example is needed; the definitions for the ICE object is probably a good test case.

The new interface

```
    callback RTCStatsCallback = void (RTCStatsReport statsReport);

    interface RTCStatsReport {
      sequence<DOMString> objects(DOMString type?);
      RTCStatsObject get(DOMString id);
    }

    interface RTCStatsObject {
        readonly attribute long timestamp;
        readonly DOMString type;
        readonly DOMString id;
        sequence<DOMString> names();
        any getValue (DOMString statName); // Primitives or sequence<primitive>
    };
```

The “type” is the type of the object being reported on. Values are registered in an IANA registry.

The “id” is a stable, unique identifier for the object being reported on. It is not intended for standardization, each implementation can have their own naming scheme. Random numbers and “type:index” are equally valid identifiers.

Documentation Format for RTCStats Objects

In order to have a well known syntax for RTCStats objects, we use Dictionary declarations. This does NOT mean that an RTCStatsObject is or contains a dictionary; due to implementation issues, we still have only the names() and getValue functions as interfaces to the RTCStatsObject. But the formalism of the Dictionary declaration syntax gives us a readable way to express the objects.

For instance, this declaration:

```
 dictionary RTCStatsFoo {
    ipAddress address;
    int bytesSent;
    int? errorsHandled;
 }
```

would declare that if the “type” of an RTCStatsObject is “RTCStatsFoo”, the names “address” and “bytesSent” MUST be present, and that the name “errorsHandled” MAY be present.

Auxillary types

These type definitions are only defined to make declarations more readable. They are not present in the DOM.

```
typedef DOMString IPAddress;
typedef DOMString StatsObjectID;
typedef sequence<DOMString> StatsObjectList;
```

The ICE Model under the new interface

This model borrows heavily from the one described in the CU-RTCWEB proposal. It is described in the form of data type registrations.
All actions are left out of the objects; most attributes are represented as members, and some members are added.

Left out for the moment:

- MediaStream
- MediaStreamTrack
- MediaStreamDescription

Included:

- RTPStreamDescription -> RTPStream (corresponds to what flows with a single SSRC)
- RealtimePort -> Candidate
- RemoteRealtimePort -> Candidate
- RealtimeTransport -> RTPSession
- CodecDescription -> Codec
- CertificateInformation (left off for now)

Added:

- CandidatePair (since we need to represent variables that are bound to a port pair)

The following type represents the local end of an SSRC.

```
dictionary RTPStream {
  int ssrc;
  int? maxBandwidth;  // as specified by user, if present
  StatsObjectID? otherEndStats; // Reference to the stats signalled from our partner.
  int? targetBandwidth;  // set by congestion control or other dynamic means
}

dictionary OutgoingRTPStream : RTP Stream {
  int packetsSent;
  long bytesSent;
}

dictionary IncomingRTPStream : RTPStream {
  int packetsReceived;
  long bytesReceived;
  int packetsLost;
}

dictionary IncomingLocalRTPStream : IncomingRTPStream {
  // References to other local objects.
  StatsObjectID transport;
  StatsObjectID mediaStreamTrack;
  StatsObjectList codecs;
}

dictionary RTPSession {
  bool open;
  int maxBandwidth;
  StatsObjectID? currentPortPair;
  StatsObjectList candidatePortPairs;
  StatsObjectList containedStreams;
}

// Candidates borrow heavily from RFC 5245 ICE terminology.
dictionary Candidate {
  IPAddress ipAddress;
  int port;
  enum {‘udp’, ‘tcp’, … } transport;
  enum {...} type;
  int priority;
  DOMString componentId;
  // RelatedAddr?
  // Base?
}

dictionary CandidatePair {
  // ICE information from RFC 5245
  StatsObjectID localCandidate;
  StatsObjectID remoteCandidate;
  bool valid;
  bool nominated;
  enum {‘closed’, ‘trying’, connected’} state;
  // Statistics
  int packetsSent;
  int packetsReceived;
  long bytesSent;
  long bytesReceived;
}

dictionary Codec {
  int payloadType;
  DOMString mediaType;  // type/subtype string
  DOMString? mediaParameters;  // from the fmtp line, if any
  int clockRate;
  int channels;
}
```

Code examples

This code checks whether an outgoing audio track has packet loss enough to cause bad audio.

```
var baseline, now;
var selector = pc.localStreams[0].audioTracks[0];
```

```
pc.getStats(selector, function (stats) {
    baseline = stats;
});

// ... wait a bit
setTimeout(function () {
    pc.getStats(selector, function (stats) {
        now = stats;
        processStats();
    });
}, aBit);

function processStats() {
    // Real code would:
    // - Check that timestamp of “local stats” and “remote stats”
    //   are reasonably consistent.
    // - Sum up over all the elements rather than just accessing
    //   element zero.

    var ssrcIds = now.objects(‘RTPStream’);
    for (i = 0; i < ssrcIds.length; i++) {
        var ssrcStatsId = ssrcIds[i];
        nowState = now.get(ssrcStatsId);
        prevState = baseline.get(ssrcStatsId);
        remoteStatsId = nowstate.get(‘otherEndStats’);
        remoteNowState = now.get(remoteStatsId);
        remotePrevState = baseline.get(remoteStatsId);
        // We should also check that
        // remoteNowState.timestamp - remotePrevState.timestamp is close to the
        // same interval as nowState.timestamp - prevState.timestamp.
        if (prevstate && remotePrevState) {
           var packetsSent = nowstate.get(‘packetsSent’) - prevstate.get(‘packetsSent’);
           var packetsReceived = remoteNowState.get(‘packetsReceived’)
                - remotePrevState.get(‘packetsReceived’);
           // if fractionLost is > 0.3, we have probably found the culprit
           var fractionLost = (packetsSent - packetsReceived) / packetsSent;
        }
   }
```