BUSINESS WHITE PAPER

What To Look For In A WebRTC Testing Tool?

WebRTC testing tools aren’t born equal. Here’s what you need to remember for your own WebRTC testing (and monitoring) needs.

spearline®

Now part of CYARA

CONTENTS

About The Author

Tsahi joined the Cyara team in 2023, following the acquisition of Spearline. Prior to that, Tsahi was the Co-founder and CEO of testRTC Ltd., which was acquired by Spearline in 2021.

Tsahi Levent-Levi
Senior Director of Product Management

Over the past two decades, Tsahi has been actively involved in the development of diverse telecommunication projects, particularly those related to WebRTC, VoIP, and 3G. His experience spans across development, management, marketing and CTO positions at RADVISION and later Amdocs. His notable contributions include developing the 3G-324M protocol stack from scratch, as well as overseeing the team responsible for the development and maintenance of the H.323 protocol stack.

Tsahi's educational background includes an MSc in Computer Science and an MBA with a specialization in Entrepreneurship and Strategy. He has several patents to his name and has served as chairman for various activity groups within the IMTC, an organization focused on interoperability and multimedia standards.

For the past 10 years, Tsahi has been working as a consultant, analyst and entrepreneur on anything related to WebRTC, CPaaS and their potential for disruption; through his consulting firm BlogGeek.me.

Common Misconceptions

When developing WebRTC applications, testing is often neglected and “saved for last”. This approach typically stems from several misconceptions:

1. That no further testing is needed as the WebRTC technology is part of the web browser.
2. That the same best practices and approaches used to test websites also apply to WebRTC.
3. That the testing of commercial products and video APIs has already been taken care of when utilizing open source packages.
4. That the outsourcing vendor you use thoroughly tests the application they are developing for you.

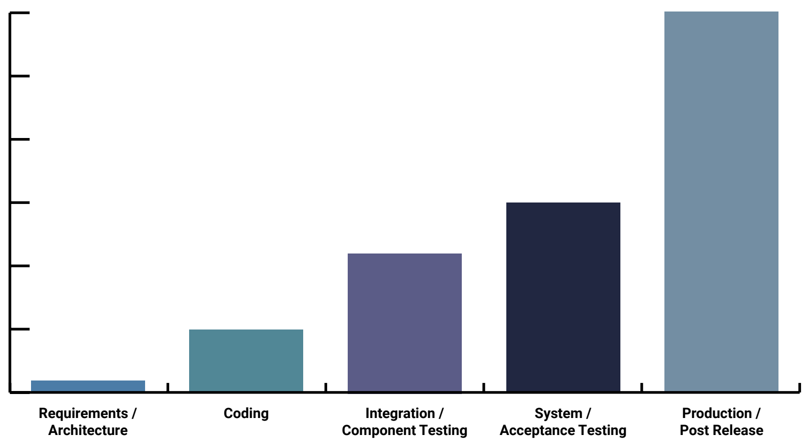
The truth? **None of the above are true.**

How do I know? Because I’ve seen it time and again with vendors requiring guidance and assistance when things break down or don’t work as advertised. Unfortunately, it is usually quite late in the process and could have been easily avoided.

You will have already seen how the cost of bugs can grow the longer they are left undetected:

With all of WebRTC’s moving parts, these bugs can be worse than usual, so finding them as early as possible is even more important.

The cost of bugs at each stage



Planning For Your WebRTC Testing

There are two main things to remember when planning your WebRTC testing:

1. Not all of your test scenarios can be covered with automated testing.

2. The more you can cover with automated testing, the better your life is going to be!

This means that you will be using multiple tools for your testing purposes. Some tools will be used to assist you with your manual testing while others will be used for your WebRTC testing automation.

For planning, go through the functional and non-functional requirements that you have. The main things you will need to collect at this stage are answers to the following questions:

1. What is the simplest test scenario you need covered?

2. How many browsers (users) can join a single session?

3. What scale do you need your service to cover? How many sessions and users will run concurrently?

4. At what point do you predict scale out on your media servers' infrastructure?

5. How do users join a session?

6. Where do your users come from geographically?

Once you have your answers for these questions, you can use the following sections to add a few more questions and requirements to your WebRTC testing plan.

The Main Actors In Every WebRTC Application

When developing and deploying a WebRTC application, there are 4 main actors:

YOUR APP

BROWSERS

NETWORKS

USER DEVICE AND PERIPHERALS

Here's the secret truth - you don't control or own most of these components. You only control your application and the rest... well, they're out of your control!

Your application - this is what you develop and put out into the world. You control and own it - how it operates, what logic it uses, etc.

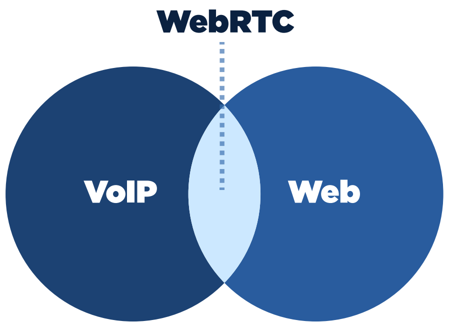
Web browsers - assuming your application runs on the web, then you don't control the browsers used to interact with your application. You can't just ask Google or Apple to hold off on the next browser upgrade to avoid breaking your application. You can't get them to implement the specific codecs you want or provide a specific behavior. You need to be able to deal with what you get and to keep up to date with the frequent browser updates.

The network - running on the open internet means the media is sent over unmanaged networks. For better or worse, you can't ask users to make calls over a good network. Well... you can, but you can't force them to - they will at some point conduct a call while driving a car or even from inside an elevator.

The user's device and peripherals - users will use whatever device they want and connect various peripherals to them. The variety of devices out there is huge, and as with networks, getting your users to choose the ones you want and have vetted is highly unlikely to happen.

Before we start, it is important to remember something: VoIP and WebRTC are different.

While WebRTC is practically VoIP, you still shouldn't be thinking that way in many cases. Why? Because of this:



WebRTC is part VoIP and part Web. As such, if you only apply the concepts, rules and tools from the VoIP world on it - you are bound to fail. And likewise, if you only apply concepts, rules and tools from the Web world on it - you will still fail. What you need is something in-between, a balanced combination that takes into account both of these worlds.

Let's see exactly what this means…

5 Critical Aspects Of Any WebRTC Testing Tool

The 4 actors in our WebRTC applications which we just discussed are going to drive a lot of what we do and how we plan with a WebRTC testing tool. Now, let's take a look at the 5 critical aspects of any WebRTC testing tool.

1. Browser Automation

At its most simple, a WebRTC testing tool relies on browser automation. To test a WebRTC application that runs on real browsers, it makes sense to also automate real browsers to validate it - this is as close as you could get to imitating real users in the real world.

Even if your application runs as a native application built on top of Google's WebRTC implementation, you would be better running most of your testing using browser automation. That's because the implementation you are going to bump into using web browsers will be quite similar to those of native applications using the same WebRTC code.

Besides taking care of browser automation, you will also need to bear in mind the upgrade frequency of browsers.

Most modern browsers get updated on a monthly cadence now. This means that each month, something may break in your application due to changes in WebRTC. Such changes are taking place so frequently and cause so many problems that this should always be a priority for you.

For browser automation, upgrading a browser will usually mean you will also need to upgrade:

- Your web browser
- WebDriver: a component used on top of browsers to enable automation
- Selenium: an open source framework for browser automation

Puppeteer can also be used instead of Selenium. While it's a different tool, it does come with similar headaches.

2. Simulating Network Behavior

WebRTC needs to have its data delivered in real time. This means really low latency is a necessity.

If you watch a Netflix or Disney+ movie, it won't matter if the media is delivered to you in a second or two, or even five. The server is likely sending you a few seconds worth of buffered content and the end of the movie will not change as you watch it…

WebRTC is different; whatever is being sent needs to be live and interactive. This means that what you have been previously taught about web testing needs to now get a WebRTC update.

We typically look at media quality in WebRTC in terms of specific media metrics: bandwidth, packet loss, jitter and latency. In general, if bandwidth is low or if the packet loss, jitter or latency is high, it is going to cause media quality to degrade.

The last mile has a lot to contribute here - the user's location is going to determine these conditions. For the last mile, we can look at 4 areas:

1. **The ISP connection** - how the user is connected to the internet
2. **The user's location in their environment** - how far they are from the access point; disturbances for reception such as being in an elevator or a basement; sharing the connection with other household members or neighbors, etc.
3. **The geographic location of the user**, in relation to the media servers and other participants in the session
4. The fact that **network conditions change dynamically**

As a WebRTC application developer, you can't just ignore these areas and attribute them to your users, but instead you need to be able to work around them and find ways to improve the perceived media quality. To do this requires a lot of testing and optimization.

Here's what you'll need in your WebRTC testing for that to happen:

- Solid and repeatable testing infrastructure. If you run the same test multiple times, it would be preferable if you could generate the same or similar results.
- The ability to run your test automation machines from multiple regions across the globe. This will let you simulate scenarios that are closer to the real world, as well as check your geolocation policies.
- Control the tested browser's network behavior. You should be able to configure its available bitrate, inject packet loss, affect jitter and latency. And you should be able to do these dynamically and programmatically in test scenarios.

3. Device Differences

Let’s move from the user’s network to their device. Here again, not all devices are born equal.

WebRTC is a significant resource hog. It requires CPU and memory to run, and usually a lot more than many other web applications would. This is due to the need to encode and decode media (we’re not even including the machine learning algorithms used for background blurring or noise suppression here).

Different devices are going to be… different. They run on different operating systems, have different CPUs with different power capabilities and they may or may not have hardware acceleration for media compression. Devices are connected to different cameras, they might use internal or external microphones or have different display resolutions.

Here’s what you can do about it:

- Have the ability to configure the virtual machines you are using in your WebRTC testing automation to simulate different types of devices.
- Get your test automation to use raw camera and microphone inputs and not pre-compressed media. This is critical to really test and evaluate your WebRTC performance.
- Manual testing. Unfortunately, there’s no way to get rid of this one for the moment. However, while doing it you will need to have a mechanism to collect all data and metrics to make it easier to debug (thankfully there’s a testRTC product for that).

4. Your Own Application’s Behavior

Your application - your code. This means you have full ownership and control of this part of your service.

We’ve already seen that testing your application needs to take place using browser automation and that the virtual machines you use need to be “big enough” to handle voice and video calls.

The reason we are using browser automation also arises from the fact that your application has a very specific workflow. It is the one you’ve already defined and decided upon. This dictates when and how users interact with your service and join the session.

But, as always, the devil is in the details. And in this case, in the fact that it “takes two to tango”.

Let’s look at two key aspects here - session and scale.

A. Synchronization Of A Single WebRTC Session

For most cases, and for the majority of test scenarios, you will need at least two users. After all, your WebRTC service is about humans interacting… so unlike most other browser automation solutions, you will need a solution that has similar out of band interaction between the automated browsers.

What does that mean? Say your service is an education service. The teacher needs to join the room before the students join. Students will need to enter a waiting room and they can then join the virtual class once the teacher is present.

How do you go about scripting these rules into your browser automation? How can your test environment know that the teacher arrived, so as to check with the student's browser that they were moved from the waiting room to the classroom?

Even worse… say your service creates an ad-hoc meeting link for the first user, which is sent out without context to the other participants. How do the other participants with the new URL join the original room?

That's the thing about "taking two to tango":

- You need messages passing between the browsers in a test to be able to send such synchronization messages and wait for them.
- Bonus point if these can also be used to check the effects on media behavior on one browser due to the actions taken on another browser in the same call.
- Another bonus point if you can create expectations and assertions on virtually any WebRTC media metric and on different slices of your test period (it is one thing to test frames per second or audio level averages for the whole scenario, but it is another thing to do it to validate that muting and unmuting channels work properly).

B. Scaling A Test Scenario

Once you nail down the single session scenario, it is time to think about scaling it up.

A single WebRTC session today can have just one user or two, but it could also have hundreds or even thousands of participants.

And then there's the fact that you'd also like to test how multiple sessions in parallel behave.

Running a Selenium Grid is straightforward these days, but doing it across multiple regions, while handling messages passing between browsers and getting all these Selenium machines to be individually controlled for network conditions - that's trickier.

Sprinkle on the need to ask (or beg) for quotas from your favorite cloud vendor (you'd be surprised how unlimited cloud resources become limited a week prior to Black Friday for example), and you have the perfect recipe for a real headache.

- Make sure your testing infrastructure is capable of scaling properly and on demand, with as little interaction from ops engineers as possible (you'll thank me later!).
- Check which test features and capabilities are "sacrificed" while running at scale - the logs and messages you won't be able to collect with your tools at that point.
- See what kind of visualizations and drilldowns are available for you at scale. You don't want to find yourself staring at endless CSV files looking for the culprit which is causing that bug in a large stress test.

5. Visibility And Repeatability

We've already examined the actors in WebRTC applications and how they affect the user experience, along with what we need to do in order to test and monitor them properly. But there's an elephant in the room that needs to be addressed as well. This one is called **visibility and repeatability**.

If you can test an issue to find a certain behavior, but you can't really reproduce it, then it has no value for you.

Once an issue is found by a monitor, if you can't visualize the results, it's going to be challenging for you to debug.

What does that mean exactly?

Having a solution that can't be reliably executed on demand yielding similar enough results will cause more problems than help. Once a bug is flagged and moved to the developers - how can developers figure out what's going on if they can't rerun the scenario causing the issue? How would the QA team test to validate that the fix provided by the developers really solves the problem?

Setting up a testing environment that can run tests, but doesn't collect the data you need or which doesn't visualize the data it collects means that you will be spending a ton of time trying to figure out what is going wrong and why. The tools you are looking for should be those that point you as quickly as possible towards the problem and from there assist in figuring out the root cause.

Here's what you should be looking for:

- Collect, analyze and visualize EVERYTHING that you can with and around WebRTC.
- To be more specific, you should collect WebRTC statistics (via getstats), WebRTC API calls and events, machine performance, browser logs, custom events, etc.
- Bonus point if there's an easy way to compare scenarios - look at results across tests to see how performance has changed between them for important KPIs. testRTC has a performance dashboard just for that.

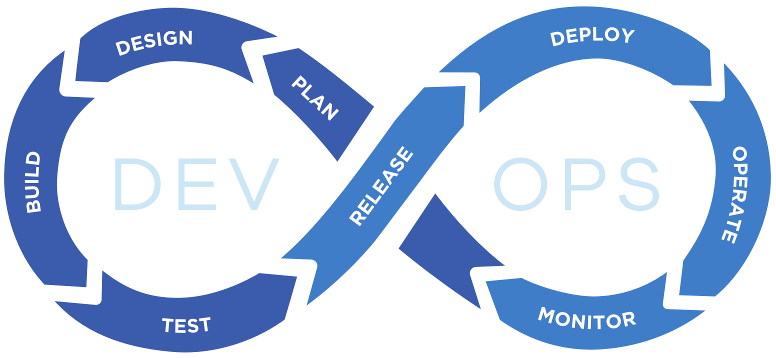
Be sure to factor manual testing into your test plans.

Manual Testing

Here is something that has to be said - no matter the technical solution(s) you are going to go with - you will still need to rely on manual testing as well.

WebRTC is so varied in the devices and networks it operates on that testing everything all the time is simply not feasible. At least not yet! This means that you'll either need to decide not to test some things at all (that 1,000,000 browsers test you always dreamed about for your startup app) or you'll need to make do with manual testing.

Monitoring and Optimization



Here's something we both know - testing is just the beginning of the journey.

Once we're "done" testing and we deploy our application, we need to maintain it. This includes monitoring, fixing issues for users, optimizing it, making sure it works with upcoming browser releases, etc.

We did testing on day one. Now, how are we going to monitor our application and optimize it on day two?

Are we going to build yet another set of tools and technologies for monitoring? Should we use different commercial solutions from different vendors for our testing and monitoring? This would mean multiple invoices… multiple signup forms… multiple dashboards to learn… multiple learning curves to go through… multiple support teams to reach out to…?

Having a "single throat to choke" for your testing AND monitoring needs is a huge advantage.

Did I mention that we offer testing, monitoring and support related solutions for your WebRTC applications?

Is testRTC (and testingRTC) The Right Tool For You?

testRTC is suitable for all of your WebRTC application lifecycle needs.

testRTC has a variety of “target customers”, from developers and testers, through to IT operations teams and support organizations. Each individual will be able to find the correct set of testRTC product that can assist them with their daily work - be it automation and scale testing, providing visibility towards the WebRTC production infrastructure or assisting in solving user complaints.

At the end of the day, testRTC has a rich and powerful set of products that are suitable for your needs if you are using WebRTC.

Only testRTC offers complete monitoring, testing and support capabilities

There are other tools out there: commercial or open source, for testing or for monitoring. Some are used to monitor users in production environments, while others might be used for test automation.

By contrast, testRTC offers a rich suite of tools:

- With testingRTC you can conduct WebRTC testing automation and do functional tests, stress tests and even use it for performance optimizations.
- upRTC lets you conduct active monitoring of your WebRTC infrastructure, while watchRTC is there to passively monitor your WebRTC user experience in production.
- qualityRTC and probeRTC together make it possible to diagnose and troubleshoot with ease connectivity issues faced by users - something that is becoming more and more important in today’s remote working environment.

testRTC provides the tools you need for the full WebRTC application lifecycle - from development - through deployment and monitoring - to support. We simulate traffic, capture live traffic, analyze neworks and visualize results in ways that make it easier for you regardless of what point in time you are with your WebRTC deployment.

The Devil is in The Details

At first glance these other alternatives and testRTC may seem similar. But take a closer look and you'll see that they can't be more different: testRTC offers a rich set of solutions covering the whole WebRTC application lifecycle.

testRTC is the first and only solution that lets you test, monitor AND support your WebRTC application. Come check us out!

The Cyara CX Transformation Platform

CRUNCHER	VOICE ASSURE	VELOCITY	PULSE	BOTIUMAI	RESOLVEAX	CENTRACX
Performance Testing	Global Telecom Assurance	Functional & Regression Testing	Customer Experience Monitoring	Conversational AI Assurance	Agent Experience Assurance	Voice of the Customer

Cyara revolutionizes the way businesses transform and optimize their customer experiences. Cyara's AI-based CX Transformation Platform empowers enterprises to deliver flawless interactions across voice, video, digital, and chatbot experiences. With Cyara, businesses improve customer journeys through continuous innovation while reducing cost and minimizing risk. With a 96% customer retention rate and world-class Net Promoter Score (NPS), today's leading global brands trust Cyara every day to deliver customer smiles at scale.

To learn more, visit cyara.com or call 1-888-GO-CYARA.

CYARA

Customer Smiles. **Delivered at Scale.**

Learn more at www.cyara.com

LinkedIn.com/company/Cyara

Twitter.com/GetCyara

YouTube.com/Cyara