Developer's Guide

Using Pacing Information to Serve Reactive Requests

4/23/2025

Contents

Using Pacing Information to Serve Reactive Requests

General information about Pacing Algorithms

The Web Engagement pacing component is designed to predict the number of media interactions that should be proactively generated by the Web Engagement Server in each succeeding time interval. For more information about pacing, consult this article.

Web Engagement also supports dual pacing, in which the pacing algorithm is able to determine how much of its capacity should be set aside in order to handle reactive traffic without allowing the proactive traffic to exceed the desired range.

In order to work with dual pacing, you should understand that:

- The pacing component works with a set of Agent Groups.
- The term *Channel* refers to a set of Agent Groups in which each group of agents is configured to work on the same, specific media channel, such as chat, web callback, or Web RTC.
- The pacing component makes predictions for each Agent Group separately by creating a dedicated thread for each Agent Group and running an instance of the pacing algorithm in each one.
- The pacing algorithm is executed at the frequency specified by the refreshPeriod option in the [pacing] section.
- The pacing algorithms used for each Agent Group monitored by the pacing component are identically configured.
- In addition to group-based predictions, the pacing component also calculates consolidated results for every channel—that is, the sum of the results for all groups belonging to a particular channel.
- There are two types of workflows:
  - **Proactive**—in which a media interaction is created every time a visitor accepts a proactively generated invitation (that is, an invitation that was triggered by specific rules associated with the Web Engagement software). With a proactive workflow, Web Engagement has complete control over when and if a given interaction is created.
  - **Reactive**—in which media interactions are created as a result of a visitor's reaction to static elements on the website, such as clicking a button or following a link. This kind of workflow is beyond the control of the Web Engagement software, since it can't control the behavior of the people who visit the site.

  Note that both proactive and reactive workflows produce the same kinds of media interactions, such as chat or callback interactions. But from the standpoint of the pacing component, proactively and reactively generated interactions have vastly different implications.

- When the pacing component is configured to calculate information for both proactive and reactive workflows, we say that it is in *dual mode* and that it has been configured to use a *dual pacing algorithm*.

- **Proactive** workflow predictions can be calculated in *both* the simple proactive mode *and* in dual mode. But **Reactive** predictions are *only* calculated in dual mode.
- The pacing component cannot distinguish between Agent Groups that have been configured to service proactive workflows and ones that are servicing reactive workflows. This distinction is completely controlled by your Genesys configuration, including the way your strategies are configured.
- The pacing component assumes that each agent it is monitoring only belongs to one of the Agent Groups it is monitoring.
- You can set up an environment where an Agent Group is configured to work with several interaction types (or channels) simultaneously. This is known as *blended* mode. In blended mode, the pacing component executes a dedicated instance of the pacing algorithm for each channel that is configured for a particular Agent Group.
- The pacing algorithms use statistical information obtained both from Stat Server and from the Web Engagement software, which has access to information that can't be obtained from Stat Server, such as the pending invitation count and the average time it takes to obtain a disposition code for an invitation.

Configuring dual pacing mode

You can specify which type of pacing algorithm to use by setting the algorithm option in the [pacing] section. This option supports the following values:

- **SUPER_PROGRESSIVE**—The Super Progressive optimization method only affects the Abandonment Rate parameter and provides a higher Busy Factor then the Predictive one. It is efficient for relatively small agent groups (1 to 30 agents) when the Predictive method gives poor results.
- **PREDICTIVE_B**—A Predictive method based on the Erlang-B queuing model. Recommended for large agent groups (more than 30 agents) with impatient customers who cannot stay in the queue, even for a short time.
- **SUPER_PROGRESSIVE_DUAL**—An adaptation of the Super Progressive method for environments serving both proactive and reactive interactions.
- **PREDICTIVE_B_DUAL**—An adaptation of the Predictive B method for environments serving both proactive and reactive interactions.

As you can see, you must specify either **SUPER_PROGRESSIVE_DUAL** or **PREDICTIVE_B_DUAL** if you want to use a dual pacing algorithm.

The most important parameter calculated by a simple pacing algorithm is called **InteractionsToSend**. This parameter determines how many proactive invitations should be sent during each refresh period. When you use a dual pacing algorithm, you need to set a balance between the percentage of agents in each group who are handling proactive invitations and those who are handling reactive ones. Without doing this, you run the risk of having your reactive traffic take over, meaning that proactively created hot leads—people who are likely to be prime customers—may be displaced by random visitors about whom you know nothing.

You can use the proactiveRatio option to adjust this balance.

Web Engagement helps avoid this issue by calculating the **InboundPortion** parameter, which specifies how much capacity should be set aside for inbound (reactive) traffic. The calculated values for **InboundPortion** can range from 0 to 1:

- 0 means that the affected page should not allow inbound traffic (for example, by disabling chat request buttons). This value will be returned by the pacing algorithm in situations where each new reactive chat request "seizes" an agent who could potentially handle a proactive chat session, thereby making it impossible to serve proactive traffic.
- 1 means that there are enough agents to serve the predicted count of proactive invitations, even if reactive interactions are started on the affected page.
- A value between 0 and 1 means that if a reactive interaction is started on the affected page, then it can potentially seize an agent who would otherwise be serving a predicted proactive interaction. This situation may be undesirable, especially if the potential value of your proactive interactions is high. In that case, you probably want to suppress the calculation of **InboundPortion**.

Suppressing Calculation of InboundPortion

Web Engagement provides two ways to suppress the calculation of **InboundPortion**:

- Use a simple, proactive-only pacing algorithm. In this case, **InboundPortion** will not be calculated at all.
- Use a dual pacing algorithm, but specify proactiveRatio at 100. In this case, the value of **InboundPortion** will always be 0, meaning that the affected page is instructed to block all inbound chat traffic, if possible—for example, by disabling chat request buttons.

The Pacing REST API

For times when reactive chats can only be controlled from the page, Web Engagement provides a RESTful Pacing API that gives you access to the value of the **InboundPortion** parameter calculated by the dual pacing algorithm.

You can also use the Pacing API to access statistical information about agent availability in the monitored Agent Groups. Although this statistical information is provided in a raw format that is used as input by the pacing algorithm, it can sometimes be critical for your understanding of how to control activity from the affected page.

Obtaining the Reactive State

Reactive state is another term that is used when talking about the **InboundPortion** parameter described above.

You can query the reactive state by issuing this request:

```
http://<gweserver.host:gweserver.port>/server/data/pacing/
reactiveState?channel=<channelName>&groups=[<names>]
```

The information returned by this request helps you understand whether reactive traffic is displacing proactive traffic on the specified channel for the specified Agent Group. If an Agent Group is not specified, the result will be calculated for the entire channel.

The response to this request is a float between 0 and 1 that indicates the probability with which the affected page should allow reactive interaction:

- **1**—There are no limitations on the number of reactive interactions.
- **0**—The page should not allow any reactive interactions.
- If the value is between `0` and `1`, the page should use the specified probability to determine whether to allow a given reactive interaction.

Let's consider an example of this last situation. If the **reactiveState** request returns a value of `0.7`, this means that you probably only want 7 out of 10 of your recently loaded pages to allow reactive interactions. Therefore, the other 3 pages should prohibit them. If you don't set up this kind of scenario, newly created reactive interactions can spiral out of control, meaning that some of them will seize agents who should have been left available for proactive customers. This means that Web Engagement will produce failed hot leads.

In JavaScript you can issue a **reactiveState** call like this:

```
<script>
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/reactiveState?channel=chat'})
    .done(function( result ) {
        console.log('result: ' + result.reactiveState);
        var rndValue = Math.random();
        if(rndValue > result.reactiveState) {
            // Disable reactive chat buttons
        }
        else {
            // Enable reactive chat buttons
        }
    });
</script>
```

Here's a sample:

```
http://example.com:9081/server/data/pacing/
reactiveState?channel=chat&groups=Web%20Engagement%20Chat
```

And the response:

```
{"reactiveState":1.0}
```

Note: This example uses the jQuery JavaScript library, which requires that jQuery be loaded on the page.

Obtaining Channel Capacity

You can use the **channelCapacity** method to understand how many concurrent interactions to allow on a specific channel for a specific Agent Group (or for the specified channel only, if a group is not explicitly specified).

Important: This method takes into account both agent state and the capacity rules that have been configured for each agent. For example, if the channel contains 1 Ready agent with a capacity of 2 and 1 Ready agent with a capacity of 3, then the cumulative channel capacity will be calculated as 5.

Important: An **InboundPortion** value of 1 does not always mean that a reactive chat will be immediately delivered to an agent.

Let's consider a situation where no agents are ready in the system and the proactive traffic is predicted at 0. This means that the value of **InboundPortion** will be 1 (because there isn't any proactive traffic to displace). However, because none of our agents are ready, you also don't want to allow any immediate reactive interactions.

By issuing a channel capacity request, you can get more information on whether or not you have to allow new reactive interactions.

Here's how to call the method:

```
http://<gweserver.host:gweserver.port>/server/data/pacing/
channelCapacity?channel=<channelName>&groups=[<names>]
```

And here is an example of how to use it in a script:

```
<script>
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/channelCapacity?channel=chat'})
    .done(function( result ) {
        console.log('Chat channel capacity is: ' + result.capacity);
    });
</script>
```

This request:

```
http://example.com:9081/server/data/pacing/
channelCapacity?channel=chat&groups=Web%20Engagement%20Chat
```

Might yield this response:

```
{"capacity":254}
```

Note: The channel capacity request provides information about the current state of channel. But you need to keep in mind the potential for race conditions.

For example, if ten browsers have requested the channel capacity concurrently, each of them could be told that the value is 1. By itself, this would lead each browser session to think that it can trigger a reactive interaction. But if an interaction is triggered on more than one browser, you will have a race condition in which the first interaction to seize an agent will use up all of the available capacity, and all other interactions will be in a wait state.

Note: This example uses the jQuery JavaScript library, which requires that jQuery be loaded on the page.

Step-by-Step Examples

Let's consider an example of how to use pacing information to determine how to serve reactive chats.

There are 2 use cases:

- The page makes sure that proactive traffic is not displaced.
- The page is not aware of proactive traffic and is interested only whether any agents are Ready.

Making sure that proactive traffic is not displaced

This is the most general use case, in which you need to avoid two different pitfalls:

- Reactive interactions should not be allowed to displace potential proactive interactions (which are calculated based on the result of the **reactiveState** method)
- Reactive interactions should only be triggered when at least one Ready agent is available on the channel

Here is the algorithm for this situation:

1. Determine whether reactive interactions are undesirable. If so, disable the request buttons on the page.
2. If reactive interactions are allowable, find out whether there are any available agents.
3. If no agents are available, disable the request buttons on the page.
4. If one or more agents are available, make sure the request buttons are enabled.

And here is a JavaScript sample:

```
function reactiveChatPacing() {
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/reactiveState?channel=chat'})
        .done(function (reactiveResult) {
            var rndValue = Math.random();

            // Check that reactive chat is allowed with probability result.reactiveState
            if (rndValue >= reactiveResult.reactiveState) {
                disableReactiveChatButtons();
            } else {

                // For the case result.reactiveState == 1 we should check channel capacity
                // as there is no guarantee that there are Ready agents
                if (reactiveResult.reactiveState == 1) {

                    $.ajax({url: 'http://{server}:{port}/server/data/pacing/
channelCapacity?channel=chat'})
                        .done(function( capacityResult ) {
                            if (capacityResult.capacity == 0) {
                                disableReactiveChatButtons();
                            } else {
                                enableReactiveChatButtons();
                            }
                        });

                }
                else {
                    enableReactiveChatButtons();
                }
            }
        });
}


function disableReactiveChatButtons () {
    // Disable reactive chat buttons
}
```

```
function enableReactiveChatButtons() {
    // Enable reactive chat buttons
}
```

Note: This example uses the jQuery JavaScript library, which requires that jQuery be loaded on the page.

Ignoring proactive traffic

This case is a shorter variant of the first one, since you only need to determine the channel capacity.

Note that you should reserve the use of this approach for situations in which you only want to support reactive interactions.

Here is the algorithm:

1. Find out whether any agents are available.

And the JavaScript:

```
function reactiveChatChannelCapacity() {
    $.ajax({url: 'http://{server}:{port}/server/data/pacing/channelCapacity?channel=chat'})
        .done(function (capacityResult) {
            if (capacityResult.capacity == 0) {
                disableReactiveChatButtons();
            } else {
                enableReactiveChatButtons();
            }
         });
}

function disableReactiveChatButtons () {
    // Disable reactive chat buttons
}

function enableReactiveChatButtons() {
    // Enable reactive chat buttons
}
```

Note: This example uses the jQuery JavaScript library, which requires that jQuery be loaded on the page.

Some Sample Calculations

70% Proactive Traffic, 30% Reactive Traffic

1. First, set your configuration options like this:
    - refreshPeriod = 2 (default value)

- proactiveRatio = `70`
- optimizationGoal = 3 (default value)
- optimizationTarget = `ABANDONMENT_RATE` (default value)
- algorithm = `SUPER_PROGRESSIVE_DUAL`

2. Then get the **InboundPortion** value by using the corresponding HTTP request on the browser side.
3. If **InboundPortion** is 1, check the channel capacity.
4. Either reduce or increase the reactive traffic, or leave it alone—depending on the result of your request, as shown in the above example script.

30% Proactive Traffic, 70% Reactive Traffic

1. First, set your configuration options like this:
   - refreshPeriod = 2 (default value)
   - proactiveRatio = `30`
   - optimizationGoal = 3 (default value)
   - optimizationTarget = `ABANDONMENT_RATE` (default value)
   - algorithm = `PREDICTIVE_B_DUAL`
2. Then get the **InboundPortion** value by using the corresponding HTTP request on the browser side.
3. If **InboundPortion** is 1, check the channel capacity.
4. Either reduce or increase the reactive traffic, or leave it alone—depending on the result of your request, as shown in the above example script.

Disable Reactive Traffic

That is, provide 100% proactive traffic by disabling all reactive chats.

1. First, set your configuration options like this:
   - refreshPeriod = 2 (default value)
   - proactiveRatio = `100`
   - optimizationGoal = 3 (default value)
   - optimizationTarget = `ABANDONMENT_RATE` (default value)
   - algorithm = `SUPER_PROGRESSIVE_DUAL`
2. Then get the **InboundPortion** value by using the corresponding HTTP request on the browser side (it must be 0).
3. Deny reactive traffic by disabling your chat buttons.

Disable Proactive Traffic

Provide 100% reactive traffic.

1. First, set your configuration options like this:
   - refreshPeriod = 2 (default value)
   - proactiveRatio = `0`
   - optimizationGoal = 3 (default value)
   - optimizationTarget = `ABANDONMENT_RATE` (default value)
   - algorithm = `SUPER_PROGRESSIVE_DUAL`
2. Then get the **InboundPortion** value by using the corresponding HTTP request on the browser side (it must be 100).
3. Allow reactive traffic by enabling your chat buttons.