Comprehensive Guide for Session Timeout and Reconnection Behavior in User Load Modeling

In today's digital landscape, ensuring robust application performance under varying user loads is paramount. One critical aspect often overlooked is handling **session timeouts** and **reconnection behaviors**. Properly modeling these scenarios in your load tests not only enhances the accuracy of your performance assessments but also ensures a seamless user experience. This guide delves into the intricacies of incorporating session timeout and reconnection behavior into user load modeling, complete with practical examples, use cases, configurations, and scenario prioritization strategies.

Table of Contents

---

1. Introduction

Load testing simulates real-world usage of applications to identify performance bottlenecks and ensure system reliability. A critical component of accurate load modeling is accounting for how users interact with session management mechanisms, particularly **session timeouts** and subsequent **reconnection behaviors**. Ignoring these factors can lead to unrealistic test scenarios, resulting in performance issues when deployed to production environments.

2. Understanding Session Timeout and Reconnection Behavior

Session Timeout

A **session timeout** occurs when a user's session expires due to inactivity or after a predetermined duration set by the server. This is a common security measure to prevent unauthorized access.

Key Points:

- **Inactivity-Based:** Triggered after a period of no user activity.
- **Absolute Timeout:** Occurs after a set duration, regardless of user activity.
- **Impact:** Users may be required to re-authenticate, potentially losing unsaved data.

Reconnection Behavior

Reconnection behavior refers to how users re-establish their session after a timeout. This can vary based on the application's design and user expectations.

Common Reconnection Strategies:

- **Re-login:** Users manually log back in.
- **Token Refresh:** Automatic renewal of authentication tokens without user intervention.
- **Page Reload:** Reloading the page to re-establish the session.

3. Use Cases

Understanding diverse scenarios where session management plays a pivotal role helps in crafting relevant load tests.

a. E-commerce Websites

- **Scenario:** Users browse products, add items to the cart, and proceed to checkout. Sessions might time out during extended browsing or while filling out lengthy forms.
- **Impact:** Session timeouts can lead to lost cart data, requiring users to re-login or start the shopping process anew, adversely affecting conversion rates.

b. SaaS Applications

- **Scenario:** Users work on projects or documents for extended periods. Inactivity might trigger session timeouts for security.
- **Impact:** Frequent timeouts disrupt workflows, leading to frustration and decreased productivity.

c. Online Banking

- **Scenario:** Users perform sensitive transactions requiring authentication. Sessions may timeout after inactivity to enhance security.
- **Impact:** Proper handling is crucial to maintain security without compromising user experience.

d. Healthcare Portals

- **Scenario:** Healthcare professionals access patient records. Sessions may timeout to protect sensitive information.
- **Impact:** Efficient reconnection mechanisms ensure continuous access without delays, vital for critical operations.

---

4. Scenario Prioritization

Not all scenarios carry the same weight in load testing. Prioritizing them ensures that critical aspects receive appropriate attention.

a. Business Impact

- **High Impact:** Scenarios that directly affect revenue or user retention (e.g., checkout process in e-commerce).
- **Medium Impact:** Features that enhance user experience but are not directly tied to revenue (e.g., profile updates).
- **Low Impact:** Auxiliary features with minimal effect on user satisfaction (e.g., viewing order history).

b. Frequency of Use

- **Frequent Use:** Commonly accessed features should be prioritized to ensure they perform reliably under load.
- **Occasional Use:** Less frequently used features are secondary but still important.

- **Rare Use:** Features rarely accessed can be tested with lower priority or during specific test phases.

c. Complexity and Risk

- **High Complexity/Risk:** Features with intricate workflows or those critical to security (e.g., authentication mechanisms).
- **Low Complexity/Risk:** Simple features with straightforward interactions.

Prioritization Matrix Example:

Scenario	Business Impact	Frequency	Complexity/Risk	Priority
Checkout Process	High	High	High	1
Browsing Products	High	High	Medium	2
Profile Updates	Medium	Medium	Low	3
Viewing Order History	Low	Low	Low	4

---

5. Determining Required User Load

Accurately determining the user load is essential to simulate real-world scenarios effectively.

a. Analyzing Traffic Patterns

- **Peak Load:** The maximum number of users the system is expected to handle concurrently.
- **Average Load:** The typical number of users interacting with the system.

Data Sources:

- **Historical Data:** Previous traffic logs.
- **Business Projections:** Expected growth or marketing campaign impacts.
- **Benchmarking:** Industry standards and competitor analysis.

b. User Behavior Modeling

Incorporate realistic user behaviors such as:

- **Think Time:** Delays between actions to mimic user decision-making.
- **Session Duration:** Average time a user spends per session.
- **Concurrency Levels:** Number of simultaneous sessions or actions.

c. Load Testing Phases

- **Baseline Testing:** Establish performance benchmarks under normal conditions.
- **Stress Testing:** Determine system limits by gradually increasing the load until failure.
- **Spike Testing:** Assess system behavior under sudden, extreme load increases.
- **Endurance Testing:** Evaluate performance over extended periods to identify memory leaks or degradation.

---

6. Practical Examples and Configurations

Implementing session timeout and reconnection behaviors varies across load testing tools. Below are detailed examples using **Apache JMeter** and **LoadRunner**.

Apache JMeter

a. Scenario: E-commerce User with Session Timeout

Objective: Simulate an e-commerce user browsing products, experiencing a session timeout after 15 minutes of inactivity, and then re-logging in to continue shopping.

Test Plan Structure:

```
Test Plan
└── Thread Group
    ├── CSV Data Set Config (User Credentials)
    ├── HTTP Header Manager (Common Headers)
    ├── HTTP Request: Browse Products
    ├── Random Timer: Think Time
    ├── HTTP Request: Add to Cart
    ├── Random Timer: Think Time
    ├── Constant Timer: 15-minute Inactivity
    ├── Response Assertion: Detect Timeout (e.g., HTTP 401)
    ├── If Controller (If Timeout Detected)
    │   └── HTTP Request: Re-login
    ├── Continue with Further User Actions
```

Configuration Details:

1. **CSV Data Set Config:**

```
<CSVDataSet>
 <stringProp name="filename">users.csv</stringProp>
 <stringProp name="variableNames">username,password</stringProp>
```

```
 <boolProp name="ignoreFirstLine">false</boolProp>
 <delimiter>,</delimiter>
 <quotedData>false</quotedData>
 <recycle>true</recycle>
 <stopThread>false</stopThread>
</CSVDataSet>
```

2. **HTTP Header Manager:**

```
<HeaderManager>
 <collectionProp name="HeaderManager.headers">
  <elementProp name="" elementType="Header">
   <stringProp name="Header.name">Content-Type</stringProp>
   <stringProp name="Header.value">application/json</stringProp>
  </elementProp>
 </collectionProp>
</HeaderManager>
```

3. **HTTP Request: Browse Products:**

```
<HTTPSamplerProxy>
 <stringProp name="HTTPSampler.domain">www.example-ecommerce.com</stringProp>
 <stringProp name="HTTPSampler.path">/browse</stringProp>
 <stringProp name="HTTPSampler.method">GET</stringProp>
</HTTPSamplerProxy>
```

4. **Random Timer: Think Time (1-5 seconds):**

```
<RandomTimer>
 <stringProp name="RandomTimer.range">4000</stringProp> <!-- Max delay in ms -->
 <stringProp name="RandomTimer.delay">1000</stringProp> <!-- Min delay in ms -->
</RandomTimer>
```

5. **HTTP Request: Add to Cart:**

```
<HTTPSamplerProxy>
 <stringProp name="HTTPSampler.domain">www.example-ecommerce.com</stringProp>
 <stringProp name="HTTPSampler.path">/add-to-cart</stringProp>
 <stringProp name="HTTPSampler.method">POST</stringProp>
 <stringProp name="HTTPSampler.arguments"> <!-- Add necessary parameters --> </stringProp>
</HTTPSamplerProxy>
```

6. **Constant Timer: 15-minute Inactivity:**

```
<ConstantTimer>
 <stringProp name="ConstantTimer.delay">900000</stringProp> <!-- 15 minutes in ms -->
</ConstantTimer>
```

7. **Response Assertion: Detect Timeout:**

```
<ResponseAssertion>
 <stringProp name="Assertion.test_field">Response Code</stringProp>
 <boolProp name="Assertion.assume_success">false</boolProp>
 <intProp name="Assertion.test_type">16</intProp> <!-- Equals -->
 <collectionProp name="Asserion.test_strings">
  <stringProp name="0">401</stringProp>
 </collectionProp>
</ResponseAssertion>
```

8. **If Controller: Re-login if Timeout Detected:**

```
<IfController>
 <stringProp name="IfController.condition">${JMeterThread.last_sample_ok} == false</stringProp>
 <HTTPSamplerProxy>
  <stringProp name="HTTPSampler.domain">www.example-ecommerce.com</stringProp>
  <stringProp name="HTTPSampler.path">/login</stringProp>
  <stringProp name="HTTPSampler.method">POST</stringProp>
  <stringProp name="HTTPSampler.arguments">
   <elementProp name="username" elementType="HTTPArgument">
    <stringProp name="Argument.name">username</stringProp>
    <stringProp name="Argument.value">${username}</stringProp>
   </elementProp>
   <elementProp name="password" elementType="HTTPArgument">
    <stringProp name="Argument.name">password</stringProp>
    <stringProp name="Argument.value">${password}</stringProp>
   </elementProp>
  </stringProp>
 </HTTPSamplerProxy>
</IfController>
```

9. **Continue with Further User Actions:**

   After re-login, the script can proceed with additional actions like continuing to browse or checkout.

b. Scenario: SaaS Application with Token Refresh

Objective: Simulate a SaaS application user performing actions throughout the day, requiring periodic OAuth token refreshes.

Test Plan Structure:

```
Test Plan
└── Thread Group
    ├── HTTP Request: Obtain Initial Access Token
    ├── JSON Extractor: Extract access_token and refresh_token
    ├── HTTP Header Manager: Add Authorization Header
    ├── Loop Controller (User Actions Loop)
    │   ├── HTTP Request: Perform Authenticated Action
    │   ├── Random Timer: Think Time
    │   ├── If Controller: Check Token Expiry
    │   │   └── HTTP Request: Refresh Token
    │   ├── Update Tokens via JSON Extractor
    ├── End Loop
```

Configuration Details:

1. **HTTP Request: Obtain Initial Access Token:**

```
<HTTPSamplerProxy>
 <stringProp name="HTTPSampler.domain">api.example-saas.com</stringProp>
 <stringProp name="HTTPSampler.path">/oauth/token</stringProp>
 <stringProp name="HTTPSampler.method">POST</stringProp>
 <stringProp name="HTTPSampler.arguments">
  <elementProp name="grant_type" elementType="HTTPArgument">
   <stringProp name="Argument.name">grant_type</stringProp>
   <stringProp name="Argument.value">password</stringProp>
  </elementProp>
  <elementProp name="username" elementType="HTTPArgument">
   <stringProp name="Argument.name">username</stringProp>
   <stringProp name="Argument.value">${username}</stringProp>
  </elementProp>
  <elementProp name="password" elementType="HTTPArgument">
   <stringProp name="Argument.name">password</stringProp>
   <stringProp name="Argument.value">${password}</stringProp>
  </elementProp>
 </stringProp>
</HTTPSamplerProxy>
```

2. **JSON Extractor: Extract access_token and refresh_token:**

```
<JSONPostProcessor>
 <stringProp
name="JSONPostProcessor.referenceNames">access_token,refresh_token</stringProp>
 <stringProp
name="JSONPostProcessor.jsonPathExpressions">$.access_token,$.refresh_token</stringProp>
</JSONPostProcessor>
```

3. **HTTP Header Manager: Add Authorization Header:**

```
<HeaderManager>
 <collectionProp name="HeaderManager.headers">
  <elementProp name="" elementType="Header">
   <stringProp name="Header.name">Authorization</stringProp>
   <stringProp name="Header.value">Bearer ${access_token}</stringProp>
  </elementProp>
 </collectionProp>
</HeaderManager>
```

4. **Loop Controller (User Actions Loop):**
   - **Number of Iterations:** Represents the number of user actions performed during the test.
5. **HTTP Request: Perform Authenticated Action:**

```
<HTTPSamplerProxy>
 <stringProp name="HTTPSampler.domain">api.example-saas.com</stringProp>
 <stringProp name="HTTPSampler.path">/data</stringProp>
 <stringProp name="HTTPSampler.method">GET</stringProp>
</HTTPSamplerProxy>
```

6. **If Controller: Check Token Expiry:**
   - **Condition:** ${__time()} > ${token_expiry_time}
   - **HTTP Request: Refresh Token:**

```
<HTTPSamplerProxy>
 <stringProp name="HTTPSampler.domain">api.example-saas.com</stringProp>
 <stringProp name="HTTPSampler.path">/oauth/token</stringProp>
 <stringProp name="HTTPSampler.method">POST</stringProp>
 <stringProp name="HTTPSampler.arguments">
  <elementProp name="grant_type" elementType="HTTPArgument">
   <stringProp name="Argument.name">grant_type</stringProp>
   <stringProp name="Argument.value">refresh_token</stringProp>
  </elementProp>
  <elementProp name="refresh_token" elementType="HTTPArgument">
   <stringProp name="Argument.name">refresh_token</stringProp>
   <stringProp name="Argument.value">${refresh_token}</stringProp>
  </elementProp>
 </stringProp>
</HTTPSamplerProxy>
```

7. **JSON Extractor: Update Tokens:**

```
<JSONPostProcessor>
 <stringProp
name="JSONPostProcessor.referenceNames">access_token,refresh_token</stringProp>
 <stringProp
name="JSONPostProcessor.jsonPathExpressions">$.access_token,$.refresh_token</stringProp>
</JSONPostProcessor>
```

LoadRunner

a. Scenario: Online Banking with Session Timeout and Re-login

Objective: Simulate an online banking user performing transactions, experiencing a session timeout after 10 minutes of inactivity, and re-authenticating to continue operations.

Script Structure:

```
Action()
{
  // Initial Login
  web_custom_request("Login",
    "URL=https://www.examplebank.com/login",
    "Method=POST",
    "Body=username={username}&password={password}",
    LAST);

  // Extract Session Token (Assuming it's in a cookie or response)
  web_reg_save_param("session_token", "LB=SessionToken=", "RB=;", LAST);

  // Perform Transactions Loop
  for(int i=0; i<50; i++) {
    // Perform Transaction
    web_custom_request("Transfer Funds",
      "URL=https://www.examplebank.com/transfer",
      "Method=POST",
      "Body=from_account=12345&to_account=67890&amount=100",
      LAST);

    // Think Time
    lr_think_time(rand() % 300 + 60); // 1 to 6 minutes

    // Simulate Inactivity Leading to Timeout
    if(i == 30) { // After 30 transactions
      lr_think_time(600); // 10 minutes
    }

    // Check for Session Timeout
    web_reg_find("Text=Session Expired", LAST);

    web_custom_request("Check Balance",
      "URL=https://www.examplebank.com/balance",
      "Method=GET",
      LAST);

    if (lr_eval_string("{SessionExpired}") != NULL) {
      // Re-login
      web_custom_request("Re-login",
        "URL=https://www.examplebank.com/login",
```

```
            "Method=POST",
            "Body=username={username}&password={password}",
            LAST);
        }
    }

    return 0;
}
```

Key Components:

1. **Initial Login:** Authenticates the user and retrieves session tokens.
2. **Transactions Loop:** Simulates user performing multiple transactions with intermittent think times.
3. **Inactivity Simulation:** Introduces a prolonged think time to trigger session timeout.
4. **Session Timeout Detection:** Checks responses for "Session Expired" messages.
5. **Re-login Process:** Re-authenticates the user to continue operations.

---

7. Advanced Configuration Tips

Enhancing your load tests with advanced configurations ensures more realistic and comprehensive simulations.

a. Parameterizing Session Timeout Durations

Avoid hardcoding timeout values to introduce flexibility and variability.

Example in JMeter:

- **User Defined Variables:**

```
<UserDefinedVariables>
 <stringProp name="SESSION_TIMEOUT">900000</stringProp> <!-- 15 minutes in ms -->
</UserDefinedVariables>
```

- **Using Variable in Timer:**

```
<ConstantTimer>
 <stringProp name="ConstantTimer.delay">${SESSION_TIMEOUT}</stringProp>
</ConstantTimer>
```

b. Randomizing Reconnection Attempts

Mimic real-world scenarios where users reconnect at varying times.

Example in JMeter:

- **Gaussian Random Timer:**

```
<GaussianRandomTimer>
 <stringProp name="ConstantTimer.delay">30000</stringProp> <!-- Mean of 30 seconds -->
 <stringProp name="GaussianRandomTimer.deviation">5000</stringProp> <!-- 5 seconds deviation
-->
</GaussianRandomTimer>
```

c. Using Logic Controllers for Complex Scenarios

Implement intricate user flows using controllers like **Loop Controllers**, **If Controllers**, and **Switch Controllers**.

Example Scenario:

- **Condition:** If session expires, attempt token refresh first.
- **Fallback:** If token refresh fails, perform a full re-login.

JMeter Test Plan Structure:

```
Test Plan
└── Thread Group
    ├── HTTP Request: Access Resource
    ├── Response Assertion: Check for Unauthorized
    ├── If Controller: If Unauthorized
    │   ├── HTTP Request: Refresh Token
    │   ├── Response Assertion: Check Refresh Success
    │   ├── If Controller: If Refresh Failed
    │   │   └── HTTP Request: Re-login
    ├── Continue with Resource Access
```

d. Implementing Custom Logic with Scripting

Leverage scripting languages (e.g., **Groovy** in JMeter) to handle dynamic session management.

Example JSR223 Sampler in JMeter (Groovy):

```
if (!prev.isSuccessful()) {
  // Log the timeout event
  log.info("Session expired. Initiating re-login.")

  // Perform re-login
  sampler = ctx.getCurrentSampler()
  loginSampler = ctx.getEngine().getTestPlan().getThreadGroups()[0].getSamplers()[0]
  ctx.setCurrentSampler(loginSampler)
```

```
    sampler.run()

    // Reset the context to continue with the main test
    ctx.setCurrentSampler(ctx.getCurrentSampler())
}
```

---

8. Monitoring and Analyzing Results

Effective monitoring and analysis are crucial to interpret load test outcomes accurately.

a. Key Metrics to Track

- **Session Expiry Rate:** Number of sessions that expire during the test.
- **Reconnection Success Rate:** Percentage of successful reconnection attempts.
- **Average Reconnection Time:** Time taken to re-establish the session.
- **Authentication Server Load:** Number of authentication requests per second.
- **Error Rates:** Frequency of session-related errors (e.g., 401 Unauthorized).

b. Tools and Techniques

- **JMeter Listeners:**
  - **View Results Tree:** Inspect individual requests and responses.
  - **Aggregate Report:** Summarize metrics like response times and error rates.
  - **Summary Report:** Obtain a high-level overview of test results.
- **External Visualization Tools:**
  - **Grafana:** For real-time monitoring and dashboards.
  - **Excel:** For detailed data analysis and charting.
- **Log Analysis:**
  - Ensure that session timeout and reconnection events are logged with timestamps for detailed post-test analysis.

c. Example Analysis Steps

1. **Collect Logs:** Ensure detailed logging of session-related events.
2. **Generate Reports:** Use JMeter's listeners to create aggregate and summary reports.
3. **Visualize Data:** Import results into Grafana or Excel for trend analysis.
4. **Identify Bottlenecks:** Determine if reconnection attempts are overloading authentication servers or causing latency issues.
5. **Validate Reconnection Logic:** Ensure that all session expiration scenarios are handled gracefully without significant performance degradation.

---

9. Best Practices

a. Realistic User Behavior Modeling

- **Diverse Session Durations:** Use varied session lengths to reflect different user behaviors.
- **Activity Patterns:** Some users are highly active, while others remain idle longer.

b. Isolate Session Management Testing

- **Separate Tests:** Conduct dedicated tests focusing solely on session management to understand its impact without interference from other factors.
- **Baseline Measurements:** Establish baseline performance metrics for session timeout and reconnection processes.

c. Optimize Reconnection Logic

- **Efficient Token Handling:** Ensure token refresh mechanisms are optimized to minimize latency.
- **Retry Strategies:** Implement exponential backoff strategies to handle transient failures during reconnection attempts.

d. Ensure Security During Testing

- **Protect Credentials:** Use secure methods to handle user credentials and tokens within test scripts.
- **Compliance:** Ensure load testing adheres to security policies, especially when dealing with sensitive applications like banking or healthcare.

e. Continuous Refinement

- **Update Test Scenarios:** Regularly update load models based on real user data and evolving application behaviors.
- **Iterative Testing:** Continuously perform load tests to capture the impact of application updates and infrastructure changes.

---

10. Conclusion

Incorporating session timeout and reconnection behavior into user load modeling is essential for creating realistic and effective performance tests. By understanding diverse use cases, prioritizing scenarios based on business impact, accurately determining user

load, and leveraging practical configurations in tools like Apache JMeter and LoadRunner, you can ensure your application gracefully handles session expirations and maintains a robust user experience under varying loads.

Key Takeaways:

- **Holistic Approach:** Consider both session management and user behavior in load testing.
- **Realism:** Simulate real-world scenarios to uncover potential performance issues.
- **Flexibility:** Use parameterization and scripting to handle dynamic session management.
- **Comprehensive Analysis:** Monitor key metrics and analyze results to identify and address bottlenecks.
- **Best Practices:** Adhere to best practices to enhance test accuracy and maintain security.

By following this guide, you can enhance your load testing strategies, ensuring your applications remain resilient, secure, and user-friendly even under peak loads.