8. Rendimiento

Una onza de rendimiento vale libras de promesas.
—Mae West

Ya es hora.

Rendimiento, es decir: se trata del tiempo y la capacidad del sistema de software para cumplir con los requisitos de tiempo. Cuando ocurren eventos (interrupciones, mensajes, solicitudes de usuarios u otros sistemas, o eventos de reloj que marcan el paso del tiempo), el sistema, o algún elemento del sistema, debe responder a ellos a tiempo. La esencia de la discusión del rendimiento es caracterizar los eventos que pueden ocurrir (y cuándo pueden ocurrir) y la respuesta basada en el tiempo del sistema o elemento a esos eventos.

Los eventos del sistema basados en la Web vienen en forma de solicitudes de los usuarios (con una numeración de decenas o decenas de millones) a través de sus clientes, como los navegadores web. En un sistema de control para un motor de combustión interna, los eventos provienen de los controles del operador y del paso del tiempo; El sistema debe controlar tanto el encendido del encendido cuando un cilindro está en la posición correcta como la mezcla de combustible para maximizar la potencia y la eficiencia y minimizar la contaminación.

Para un sistema basado en la web, la respuesta deseada puede expresarse como la cantidad de transacciones que se pueden procesar en un minuto. Para el sistema de control del motor, la respuesta podría ser la variación permitida en el tiempo de encendido. En cada caso, el patrón de eventos que llegan y el patrón de respuestas se pueden caracterizar, y esta caracterización forma el lenguaje con el cual se construyen escenarios de desempeño.

Durante gran parte de la historia de la ingeniería de software, el rendimiento ha sido el factor determinante en la arquitectura del sistema. Como tal, con frecuencia ha comprometido el logro de todas las demás cualidades. A medida que la relación precio / rendimiento del hardware continúa cayendo en picado y el costo de desarrollar software continúa aumentando, otras cualidades han emergido como competidores importantes para el rendimiento.

Sin embargo, todos los sistemas tienen requisitos de rendimiento, incluso si no están expresados. Por ejemplo, una herramienta de procesamiento de texto puede no tener ningún requisito de rendimiento explícito, pero sin duda todos estarán de acuerdo en que esperar unUna hora (o un minuto o un segundo) antes de ver aparecer un carácter escrito en la pantalla es inaceptable. El rendimiento sigue siendo un atributo de calidad fundamental para todos los programas.

El rendimiento a menudo está vinculado a la escalabilidad, es decir, aumenta la capacidad de trabajo de su sistema, mientras sigue teniendo un buen desempeño. Técnicamente, la escalabilidad hace que su sistema sea fácil de cambiar de una manera particular, y también lo es un tipo de modificabilidad. Además, abordamos la escalabilidad explícitamente en el Capítulo 12 .

8.1. ESCENARIO GENERAL DE RENDIMIENTO

Un escenario de rendimiento comienza con un evento que llega al sistema. Responder correctamente al evento requiere recursos (incluido el tiempo) para ser consumido. Mientras esto sucede, el sistema puede atender simultáneamente otros eventos.

Concurrencia

La concurrencia es uno de los conceptos más importantes que un arquitecto debe entender y uno de los menos enseñados en los cursos de ciencias de la computación. La concurrencia se refiere a operaciones que ocurren en paralelo. Por ejemplo, supongamos que hay un hilo que ejecuta las declaraciones

```
x: = 1;
x ++;
```

y otro hilo que ejecuta las mismas sentencias. ¿Cuál es el valor de *x* después de que ambos subprocesos hayan ejecutado esas declaraciones? Podría ser 2 o 3. Te lo dejo a ti para que averigües cómo podría ocurrir el valor 3, ¿o debería decir que te lo interpuse?

La concurrencia ocurre cada vez que su sistema crea un nuevo hilo, porque los hilos, por definición, son secuencias de control independientes. La multitarea en su sistema es compatible con subprocesos independientes. Se admiten simultáneamente múltiples usuarios en su sistema mediante el uso de subprocesos. La concurrencia también se produce cada vez que su sistema se ejecuta en más de un procesador, ya sea que los procesadores se empaqueten por separado o como procesadores de múltiples núcleos. Además, debe tener en cuenta la concurrencia cuando su sistema utiliza algoritmos paralelos, infraestructuras de paralelización, tales como map-reduce, o bases de datos NoSQL, o utiliza uno de una variedad de algoritmos de programación

concurrentes. En otras palabras, la concurrencia es una herramienta disponible para usted de muchas maneras.

La concurrencia, cuando tiene varias CPU o estados de espera que pueden explotar, es una buena cosa. Permitir que las operaciones se realicen en paralelo mejora el rendimiento, ya que los retrasos introducidos en un subproceso permiten que el procesadorpara avanzar en otro hilo. Pero debido al fenómeno de intercalación que se acaba de describir (lo que se conoce como una *condición de carrera*), la concurrencia también debe ser administrada cuidadosamente por el arquitecto.

Como muestra el ejemplo, las condiciones de carrera pueden ocurrir cuando hay dos hilos de control y hay un estado compartido. La gestión de la concurrencia a menudo se reduce a administrar cómo se comparte el estado. Una técnica para prevenir las condiciones de carrera es usar bloqueos para imponer el acceso secuencial al estado. Otra técnica es dividir el estado en función del subproceso que ejecuta una parte del código. Es decir, si hay dos instancias de *x* en nuestro ejemplo, *x* no está compartida por los dos subprocesos y no habrá una condición de carrera.

Las condiciones de carrera son uno de los tipos más difíciles de descubrir; la ocurrencia del error es esporádica y depende de las diferencias (posiblemente pequeñas) en el tiempo. Una vez tuve una condición de carrera en un sistema operativo que no pude localizar. Puse una prueba en el código para que la próxima vez que ocurriera la condición de carrera, se activara un proceso de depuración. Tomó más de un año para que el error se repitiera y así se pudiera determinar la causa.

No permita que las dificultades asociadas con la concurrencia lo disuadan de utilizar esta técnica tan importante. Solo úselo sabiendo que debe identificar cuidadosamente las secciones críticas en su código y asegurarse de que no se produzcan condiciones de carrera en esas secciones.

- *LB*

Los eventos pueden llegar en patrones predecibles o distribuciones matemáticas, o ser impredecibles. Un patrón de llegada para eventos se caracteriza como *periódico* , *estocástico* o *esporádico* :

• Los eventos periódicos llegan predeciblemente a intervalos de tiempo regulares. Por ejemplo, un evento puede llegar cada 10 milisegundos. La llegada periódica de eventos es más frecuente en los sistemas en tiempo real.

• La llegada estocástica significa que los eventos llegan de acuerdo con alguna distribución probabilística.

• Los eventos esporádicos llegan según un patrón que no es ni periódico ni estocástico. Incluso estos pueden caracterizarse, sin embargo, en

ciertas circunstancias. Por ejemplo, podríamos saber que en la mayoría de los 600 eventos ocurrirán en un minuto, o que habrá al menos 200 milisegundos entre la llegada de cualquiera de los dos eventos. (Esto podría describir un sistema en el que los eventos corresponden a los golpes del teclado de un usuario humano). Estas son caracterizaciones útiles, aunque no sabemos cuándo llegará un solo evento.

La respuesta del sistema a un estímulo se puede medir de la siguiente manera:

• *Latencia*. El tiempo entre la llegada del estímulo y la respuesta del sistema a él.

• *Plazos de tramitación*. En el controlador del motor, por ejemplo, el combustible debe encenderse cuando el cilindro está en una posición particular, introduciendo así una fecha límite de procesamiento.

• El *rendimiento* del sistema, generalmente dado como el número de transacciones que el sistema puede procesar en una unidad de tiempo.

• El *jitter* de la respuesta: la variación permitida en la latencia.

• La *cantidad de eventos no procesados* porque el sistema estaba demasiado ocupado para responder.

A partir de estas consideraciones, ahora podemos describir las partes individuales de un escenario general para el rendimiento:

• *Fuente de estímulo*. Los estímulos llegan de fuentes externas (posiblemente múltiples) o internas.

• *Estímulo* . Los estímulos son las llegadas del evento. El patrón de llegada puede ser periódico, estocástico o esporádico, caracterizado por parámetros numéricos.

• *Artefacto* . El artefacto es el sistema o uno o más de sus componentes.

• *Medio ambiente* . El sistema puede estar en varios modos operativos, como normal, emergencia, carga máxima o sobrecarga.

• *Respuesta* . El sistema debe procesar los eventos que llegan. Esto puede causar un cambio en el entorno del sistema (por ejemplo, del modo normal al modo de sobrecarga).

• *Medida de respuesta*. Las medidas de respuesta son el tiempo que lleva procesar los eventos que llegan (latencia o fecha límite), la variación en este tiempo (jitter), la cantidad de eventos que se pueden procesar dentro de un intervalo de tiempo particular (rendimiento) o una caracterización de los eventos que no pueden ser procesados (tasa de error).

El escenario general de rendimiento se resume en la Tabla 8.1 .

Tabla 8.1. Escenario General de Rendimiento

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

La Figura 8.1 muestra un ejemplo de escenario de rendimiento concreto: los usuarios inician transacciones en operaciones normales. El sistema procesa las transacciones con una latencia promedio de dos segundos.

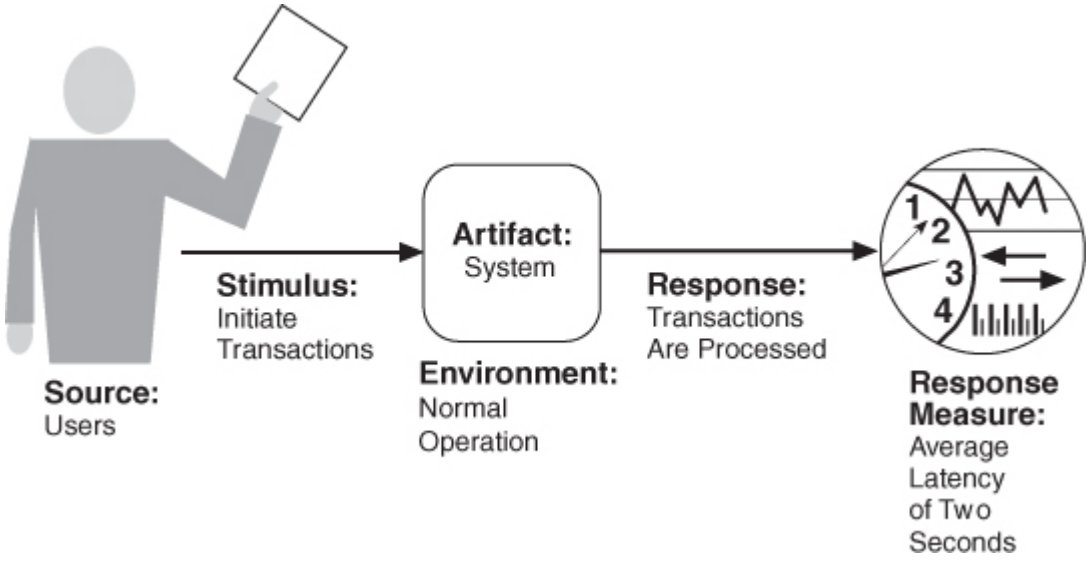

Figura 8.1. Ejemplo de escenario de rendimiento concreto.

8.2. TÁCTICAS PARA EL RENDIMIENTO

El objetivo de las tácticas de rendimiento es generar una respuesta a un evento que llega al sistema dentro de alguna restricción basada en el tiempo. El evento puede ser único o una secuencia y es el desencadenante para realizar el cálculo. Las tácticas de rendimiento controlan el tiempo dentro del cual se genera una respuesta, como se ilustra en la Figura 8.2 .

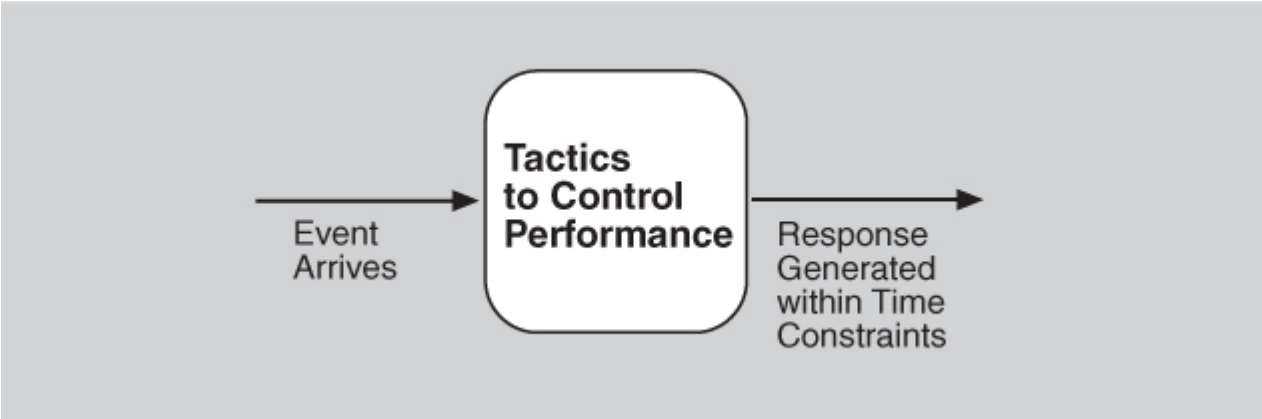

Figura 8.2. El objetivo de las tácticas de rendimiento.

En cualquier momento durante el período posterior a la llegada de un evento, pero antes de que se complete la respuesta del sistema, el sistema está trabajando para responder a ese evento o el procesamiento está bloqueado por algún motivo. Esto lleva a los dos contribuyentes básicos al tiempo de respuesta: tiempo de procesamiento (cuando el sistema está trabajando para responder) y tiempo bloqueado (cuando el sistema no puede responder).

• *Tiempo de procesamiento*. El procesamiento consume recursos, lo que lleva tiempo. Los eventos se manejan mediante la ejecución de uno o más componentes, cuyo tiempo empleado es un recurso. Los recursos de hardware incluyen CPU, almacenes de datos, ancho de banda de comunicación de red y memoria. Los recursos de software incluyen entidades definidas por el sistema bajo diseño. Por ejemplo, los buffers deben ser administrados y el acceso a las secciones críticas [1] debe ser secuencial.

Por ejemplo, supongamos que un mensaje es generado por un componente. Puede colocarse en la red, después de lo cual llega a otro componente. Luego se coloca en un búfer; transformado de alguna manera; procesado de acuerdo a algún algoritmo; transformado para la salida; colocado en un búfer de salida; y enviado a otro componente, otro sistema o algún actor. Cada uno de estos pasos consume recursos y tiempo, y contribuye a la latencia general del procesamiento de ese evento.

Los diferentes recursos se comportan de manera diferente a medida que su utilización se acerca a su capacidad, es decir, a medida que se saturan. Por ejemplo, a medida que la CPU se carga más, el rendimiento generalmente se degrada de manera constante. Por otro

lado, cuando empiezas a quedarte sin memoria, en algún momento el intercambio de páginas se vuelve abrumador y el rendimiento falla repentinamente.

• *Tiempo bloqueado*. Se puede bloquear un cálculo debido a la disputa por algún recurso necesario, porque el recurso no está disponible o porque el cálculo depende del resultado de otros cálculos que aún no están disponibles:

• *Contienda por los recursos*. Muchos recursos solo pueden ser utilizados por un solo cliente a la vez. Esto significa que otros clientes deben esperar para acceder a esos recursos. La figura 8.2 muestra los eventos que llegan al sistema. Estos eventos pueden estar en un solo flujo o en múltiples flujos. Múltiples flujos que compiten por el mismo recurso o diferentes eventos en el mismo flujo que compiten por el mismo recurso contribuyen a la latencia. Cuanta más contención haya por un recurso, mayor será la probabilidad de que se introduzca la latencia.

• *Disponibilidad de recursos*. Incluso en ausencia de contención, el cálculo no puede continuar si un recurso no está disponible. La falta de disponibilidad puede deberse a que el recurso está fuera de línea o por una falla del componente o por algún otro motivo. En cualquier caso, debe identificar los lugares donde la falta de disponibilidad de recursos podría causar una contribución significativa a la latencia general. Algunas de nuestras tácticas están destinadas a lidiar con esta situación.

• *Dependencia de otros cálculos*. Un cálculo puede tener que esperar porque debe sincronizarse con los resultados de otro cálculo o porque está esperando los resultados de un cálculo que inició. Si un componente llama a otro componente y debe esperar a que ese componente responda, el tiempo puede ser significativo si el componente al que se llama está en el otro extremo de la red (en lugar de estar ubicado en el mismo procesador).

Con estos antecedentes, pasamos a nuestras categorías tácticas. Podemos reducir la demanda de recursos o hacer que los recursos que tenemos manejen la demanda de manera más efectiva:

• *Controlar la demanda de recursos*. Esta táctica opera en el lado de la demanda para producir una menor demanda de los recursos que tendrán que atender los eventos.

• *Gestionar los recursos*. Esta táctica opera en el lado de la respuesta para hacer que los recursos disponibles trabajen de manera más efectiva para manejar las demandas que se les presentan.

Control de la demanda de recursos

Una forma de aumentar el rendimiento es administrar cuidadosamente la demanda de recursos. Esto se puede hacer reduciendo el número de eventos procesados aplicando una tasa de muestreo o limitando la velocidad a la que el sistema responde a los eventos. Además, hay una serie de técnicas para garantizar que los recursos que tiene se apliquen con criterio:

• *Gestionar frecuencia de muestreo* . Si es posible reducir la frecuencia de muestreo a la que se captura un flujo de datos ambientales, entonces se puede reducir la demanda, generalmente con alguna pérdida de fidelidad. Esto es común en los sistemas de procesamiento de señales donde, por ejemplo, se pueden elegir diferentes códecs con diferentes tasas de muestreo y formatos de datos. Esta elección de diseño se realiza para mantener niveles predecibles de latencia; debe decidir si tener una fidelidad más baja pero un flujo de datos consistente es preferible a perder paquetes de datos.

• *Limitar la respuesta al evento*. Cuando los eventos discretos llegan al sistema (o elemento) demasiado rápido para ser procesados, entonces los eventos deben ponerse en cola hasta que puedan procesarse. Debido a que estos eventos son discretos, normalmente no es deseable "submuestrearlos". En tal caso, puede elegir procesar eventos solo hasta una tasa máxima establecida, asegurando así un procesamiento más predecible cuando los eventos se procesan realmente. Esta táctica podría ser activada por un tamaño de cola o una medida de utilización del procesador que exceda algún nivel de advertencia. Si adopta esta táctica y no es aceptable perder ningún evento, debe asegurarse de que sus colas sean lo suficientemente grandes como para manejar el peor de los casos. Si, por otro lado, elige eliminar eventos, debe elegir una política para manejar esta situación: ¿Registra los eventos eliminados o simplemente los ignora? ¿Notifica a otros sistemas, usuarios,

• *Priorizar eventos* . Si no todos los eventos son igual de importantes, puede imponer un esquema de prioridad que clasifique los eventos de acuerdo con lo importante que es atenderlos. Si no hay suficientes recursos disponibles para atenderlos cuando surjan, los eventos de baja prioridad pueden ignorarse. Ignorar eventos consume recursos

mínimos (incluido el tiempo) y, por lo tanto, aumenta el rendimiento en comparación con un sistema que atiende todos los eventos todo el tiempo. Por ejemplo, un edificio.El sistema de gestión puede elevar una variedad de alarmas. Las alarmas que amenazan la vida, como las alarmas contra incendios, deben recibir mayor prioridad que las alarmas informativas, ya que una habitación está demasiado fría.

• *Reducir los gastos generales* . El uso de intermediarios (tan importante para la modificabilidad, como vimos en el Capítulo 7) aumenta los recursos consumidos en el procesamiento de un flujo de eventos, y así eliminarlos mejora la latencia. Este es un clásico intercambio de modificabilidad / rendimiento. La separación de las preocupaciones, otra pieza clave de la modificabilidad, también puede aumentar la sobrecarga de procesamiento necesaria para dar servicio a un evento si conduce a que un evento sea atendido por una cadena de componentes en lugar de un solo componente. Los costos de conmutación de contexto y de comunicación entre componentes se suman, especialmente cuando los componentes están en nodos diferentes en una red. Una estrategia para reducir la sobrecarga computacional es la ubicación conjunta de recursos. La ubicación conjunta puede significar alojar componentes que cooperan en el mismo procesador para evitar el retraso de la comunicación de la red; puede significar colocar los recursos en el mismo componente de software de tiempo de ejecución para evitar incluso el gasto de una llamada de subrutina. Un caso especial de reducción de la sobrecarga computacional es realizar una limpieza periódica de los recursos que se han vuelto ineficientes. Por ejemplo, las tablas hash y los mapas de memoria virtual pueden requerir el recálculo y la reinicialización. Otra estrategia común es ejecutar servidores de un solo hilo (para simplificar y evitar la contención) y dividir la carga de trabajo entre ellos.

• *Tiempos de ejecución consolidados* . Ponga un límite en cuánto tiempo de ejecución se utiliza para responder a un evento. Para algoritmos iterativos, dependientes de los datos, limitar el número de iteraciones es un método para limitar los tiempos de ejecución. El costo suele ser un cálculo menos preciso. Si adopta esta táctica, deberá evaluar su efecto en la precisión y ver si el resultado es "lo suficientemente bueno". Esta táctica de administración de recursos frecuentemente se combina con la táctica de administración de tasa de muestreo.

• *Aumentar la eficiencia de los recursos*. Mejorar los algoritmos utilizados en áreas críticas disminuirá la latencia.

Administrar recursos

Incluso si la demanda de recursos no es controlable, la administración de estos recursos puede ser. A veces un recurso puede ser cambiado por otro. Por ejemplo, los datos intermedios se pueden mantener en un caché o se pueden regenerar dependiendo de la disponibilidad de recursos de tiempo y espacio. Esta táctica generalmente se aplica al procesador, pero también es efectiva cuando se aplica a otros recursos, como un disco. Aquí hay algunas tácticas de gestión de recursos:

• *Aumentar los recursos*. Los procesadores más rápidos, los procesadores adicionales, la memoria adicional y las redes más rápidas tienen el potencial de reducir la latencia.El costo generalmente es una consideración en la elección de recursos, pero aumentar los recursos es definitivamente una táctica para reducir la latencia y, en muchos casos, es la forma más económica de obtener una mejora inmediata.

• *Introducir concurrencia* . Si las solicitudes se pueden procesar en paralelo, se puede reducir el tiempo bloqueado. Se puede introducir la concurrencia al procesar diferentes flujos de eventos en diferentes subprocesos o al crear subprocesos adicionales para procesar diferentes conjuntos de actividades. Una vez que se ha introducido la concurrencia, se pueden usar políticas de programación para lograr los objetivos que considere deseables. Las diferentes políticas de programación pueden maximizar la imparcialidad (todas las solicitudes obtienen el mismo tiempo), el rendimiento (el tiempo más corto para finalizar primero) u otros objetivos. (Vea la barra lateral.)

• *Mantener múltiples copias de cómputos* . Varios servidores en un patrón cliente-servidor son réplicas de cómputo. El propósito de las réplicas es reducir la contención que se produciría si todos los cálculos tuvieran lugar en un solo servidor. Un *equilibrador de carga* es una pieza de software que asigna nuevo trabajo a uno de los servidores duplicados disponibles; los criterios para la asignación varían, pero pueden ser tan simples como el turno redondo o la asignación de la siguiente solicitud al servidor menos ocupado.

• *Mantener múltiples copias de datos*. *El almacenamiento en caché* es una táctica que implica mantener copias de datos (posiblemente uno un subconjunto del otro) en el almacenamiento con diferentes velocidades de acceso. Las diferentes velocidades de acceso pueden ser

inherentes (memoria versus almacenamiento secundario) o pueden deberse a la necesidad de comunicación de la red. *Replicación de datos*implica mantener copias separadas de los datos para reducir la contención de múltiples accesos simultáneos. Debido a que los datos que se almacenan en caché o replican generalmente son una copia de los datos existentes, mantener las copias consistentes y sincronizadas se convierte en una responsabilidad que el sistema debe asumir. Otra responsabilidad es elegir los datos a almacenar en caché. Algunos cachés operan simplemente conservando copias de lo que se solicitó recientemente, pero también es posible predecir las solicitudes futuras de los usuarios según los patrones de comportamiento, y comenzar los cálculos o las consultas previas necesarias para cumplir con esas solicitudes antes de que el usuario las haya realizado.

• *Tamaños de cola enlazados* . Esto controla el número máximo de llegadas en cola y, en consecuencia, los recursos utilizados para procesar las llegadas. Si adopta esta táctica, debe adoptar una política para lo que sucede cuando se desbordan las colas y decidir si no es aceptable no responder a los eventos perdidos. Esta táctica se empareja con frecuencia con la táctica de respuesta de evento límite.

• *Programar recursos* . Siempre que haya una disputa por un recurso, el recurso debe ser programado. Los procesadores están programados, los almacenamientos intermedios están programados y las redes programadas. Su objetivo es comprender las características del uso de cada recurso y elegir la estrategia de programación que sea compatible con él. (Vea la barra lateral.)

Las tácticas para el rendimiento se resumen en la Figura 8.3 .

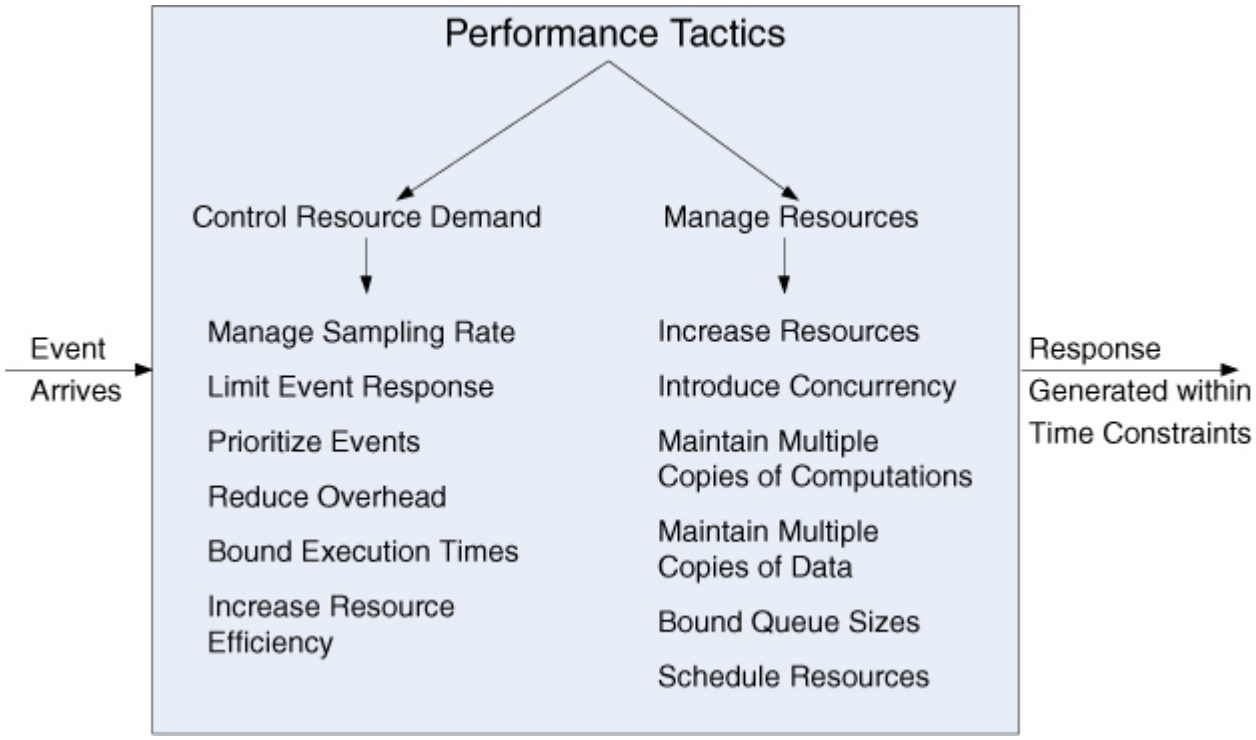

Figura 8.3. Tácticas de rendimiento

Políticas de programación

Una *política de programación* tiene dos partes: una asignación de prioridad y un envío. Todas las políticas de programación asignan prioridades. En algunos casos, la asignación es tan simple como primero en entrar / primero en salir (o FIFO). En otros casos, puede estar vinculado a la fecha límite de la solicitud o su importancia semántica. Los criterios de competencia para la programación incluyen el uso óptimo de los recursos, la importancia de la solicitud, la minimización de la cantidad de recursos utilizados, la minimización de la latencia, la maximización del rendimiento, la prevención de la inanición para garantizar la imparcialidad, etc. Debe tener en cuenta estos criterios posiblemente conflictivos y el efecto que la táctica elegida tiene para cumplirlos.

Un flujo de eventos de alta prioridad se puede enviar solo si el recurso al que se está asignando está disponible. En ocasiones, esto depende de que se anule al usuario actual del recurso. Las posibles opciones de preferencia son las siguientes: pueden ocurrir en cualquier momento, pueden ocurrir solo en puntos de preferencia específicos, y los procesos de ejecución no pueden ser anticipados. Algunas políticas comunes de programación son las siguientes:

• *Primero en entrar / primero en salir*. Las colas FIFO tratan todas las solicitudes de recursos como iguales y las satisfacen a su vez. Una posibilidad con una cola FIFO es que una solicitud se quede detrás de otra que demore mucho tiempo en generar una respuesta. Mientras todas las solicitudes sean

realmente iguales, esto no es un problema, pero si algunas solicitudes son de mayor prioridad que otras, es problemático.

• *Programación de prioridad fija*. La programación de prioridad fija asigna a cada fuente de recursos una prioridad particular y asigna los recursos en ese orden de prioridad. Esta estrategia garantiza un mejor servicio para las solicitudes de mayor prioridad. Pero admite la posibilidad de una solicitud de prioridad más baja, pero importante, que demora un tiempo arbitrariamente largo en ser atendida, porque está atascada detrás de una serie de solicitudes de prioridad más alta. Tres estrategias comunes de priorización son estas:

• *Importancia semántica*. A cada flujo se le asigna una prioridad de forma estática de acuerdo con algunas características del dominio de la tarea que lo genera.

• *Plazo monotónico*. Plazo monotónico. La fecha límite monotónica es una asignación de prioridad estática que asigna una mayor prioridad a los flujos con fechas límite más cortas. Esta política de programación se utiliza cuando se deben programar transmisiones de diferentes prioridades con fechas límite en tiempo real.

• *Tasa monotónica*. Rate monotonic es una asignación de prioridad estática para flujos periódicos que asigna mayor prioridad a flujos con períodos más cortos. Esta política de programación es un caso especial de fecha límite monotónica, pero es más conocida y es más probable que sea compatible con el sistema operativo.

• *Programación dinámica de prioridades*. Las estrategias incluyen estas:

• *Round-robin*. Round-robin es una estrategia de programación que ordena las solicitudes y luego, en cada posibilidad de asignación, asigna el recurso a la siguiente solicitud en ese orden. Una forma especial deRound-Robin es un ejecutivo cíclico, donde las posibilidades de asignación son a intervalos de tiempo fijos.

• *Primera fecha límite primero*. Primera fecha límite primero. Early-deadline-first-first asigna prioridades basadas en las solicitudes pendientes con la fecha límite más temprana.

• *Menos flojo primero*. Esta estrategia asigna la prioridad más alta al trabajo que tiene el menor "tiempo de inactividad", que es la diferencia entre el tiempo de ejecución restante y el tiempo hasta la fecha límite del trabajo.

Para un solo procesador y procesos que son preventivos (es decir, es posible suspender el procesamiento de una tarea para atender una tarea cuyo plazo se acerca), tanto la fecha límite más temprana como las estrategias de programación con menos demoras son óptimas. Es decir, si el conjunto de procesos se puede programar para que se cumplan todos los plazos, entonces estas estrategias podrán programar ese conjunto con éxito.

• *Programación estática*. Un programa ejecutivo cíclico es una estrategia de programación donde los puntos de prioridad y la secuencia de asignación al

recurso se determinan fuera de línea. De este modo, se evita la sobrecarga de tiempo de ejecución de un planificador.

Tácticas de rendimiento en la carretera

Las tácticas son principios de diseño genéricos. Para ejercer este punto, piense en el diseño de los sistemas de carreteras y autopistas donde vive. Los ingenieros de tráfico emplean un montón de "trucos" de diseño para optimizar el rendimiento de estos sistemas complejos, donde el rendimiento tiene una serie de medidas, como el rendimiento (cuántos autos por hora llegan desde los suburbios al estadio de fútbol), latencia promedio de la caja (cuánto tarda, en promedio, llegar de su casa al centro de la ciudad), y la latencia en el peor de los casos (cuánto tiempo le toma a un vehículo de emergencia llevarlo al hospital). ¿Cuáles son estos trucos? Nada menos que nuestros buenos viejos amigos, tácticas.

Consideremos algunos ejemplos:

• *Gestionar la tasa de eventos*. Las luces en las rampas de entrada a la autopista permiten que los automóviles ingresen a la autopista solo a intervalos establecidos, y los automóviles deben esperar (hacer cola) en la rampa su turno.

• *Priorizar eventos*. Las ambulancias y la policía, con sus luces y sirenas encendidas, tienen mayor prioridad que los ciudadanos comunes; algunas carreteras tienen carriles para vehículos de alta ocupación (HOV), dando prioridad a los vehículos con dos o más ocupantes.

• *Mantener múltiples copias*. Agregue carriles de tráfico a las carreteras existentes o construya rutas paralelas.

Además, hay algunos trucos que los usuarios del sistema pueden emplear:

• *Aumentar los recursos*. Compre un Ferrari, por ejemplo. En igualdad de condiciones, el automóvil más rápido con un conductor competente en una carretera abierta lo llevará a su destino más rápidamente.

• *Aumentar la eficiencia*. Encuentre una nueva ruta que sea más rápida y / o más corta que su ruta actual.

• *Reducir la sobrecarga computacional*. Puede conducir más cerca del automóvil que está frente a usted, o puede cargar más personas en el mismo vehículo (es decir, compartir el automóvil).

¿Cuál es el punto de esta discusión? Parafraseando a Gertrude Stein: el rendimiento es el rendimiento es el rendimiento. Los ingenieros han estado analizando y optimizando los sistemas durante siglos, tratando de mejorar su rendimiento, y han estado empleando las mismas estrategias de diseño para hacerlo. Por lo tanto, debería sentirse un poco más cómodo al saber que cuando intenta mejorar el rendimiento de su sistema basado en computadora, está aplicando tácticas que han sido "probadas a fondo".

- *RK*

8.3. UNA LISTA DE VERIFICACIÓN DE DISEÑO PARA EL RENDIMIENTO

La Tabla 8.2 es una lista de verificación para respaldar el proceso de diseño y análisis para el desempeño.

Tabla 8.2. Lista de verificación para apoyar el proceso de diseño y análisis para el rendimiento

Category	Checklist
Allocation of Responsibilities	Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those responsibilities, identify the processing requirements of each responsibility, and determine whether they may cause bottlenecks. Also, identify additional responsibilities to recognize and process requests appropriately, including ▪ Responsibilities that result from a thread of control crossing process or processor boundaries ▪ Responsibilities to manage the threads of control—allocation and deallocation of threads, maintaining thread pools, and so forth ▪ Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).
Coordination Model	Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that do the following: ▪ Support any introduced concurrency (for example, is it thread safe?), event prioritization, or scheduling strategy ▪ Ensure that the required performance response can be delivered ▪ Can capture periodic, stochastic, or sporadic event arrivals, as needed ▪ Have the appropriate properties of the communication mechanisms; for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency

Data Model	Determine those portions of the data model that will be heavily loaded, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those data abstractions, determine the following: ▪ Whether maintaining multiple copies of key data would benefit performance ▪ Whether partitioning data would benefit performance ▪ Whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible ▪ Whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible
Mapping among Architectural Elements	Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency. Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity. Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance. Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.
Resource Management	Determine which resources in your system are critical for performance. For these resources, ensure that they will be monitored and managed under normal and overloaded system operation. For example:

	▪ System elements that need to be aware of, and manage, time and other performance-critical resources ▪ Process/thread models ▪ Prioritization of resources and access to resources ▪ Scheduling and locking strategies ▪ Deploying additional resources on demand to meet increased loads
Binding Time	For each element that will be bound after compile time, determine the following: ▪ Time necessary to complete the binding ▪ Additional overhead introduced by using the late binding mechanism Ensure that these values do not pose unacceptable performance penalties on the system.
Choice of Technology	Will your choice of technology let you set and meet hard, real-time deadlines? Do you know its characteristics under load and its limits? Does your choice of technology give you the ability to set the following: ▪ Scheduling policy ▪ Priorities ▪ Policies for reducing demand ▪ Allocation of portions of the technology to processors ▪ Other performance-related parameters Does your choice of technology introduce excessive overhead for heavily used operations?

8.4. RESUMEN

El rendimiento se trata de la gestión de los recursos del sistema frente a tipos particulares de demanda para lograr un comportamiento de tiempo aceptable. El rendimiento se puede medir en términos de rendimiento y latencia para sistemas tanto interactivos como de tiempo real integrados, aunque el rendimiento generalmente es más importante en los sistemas interactivos, y la latencia es más importante en los sistemas integrados.

El rendimiento puede mejorarse reduciendo la demanda o administrando los recursos de manera más adecuada. Reducir la demanda tendrá el efecto secundario de reducir la fidelidad o negarse a atender algunas solicitudes. La administración más adecuada de los recursos se puede realizar a través de la programación, la replicación o simplemente aumentando los recursos disponibles.