PERFORMANCE

Rendimiento, es decir: es cuestión de tiempo y la capacidad del sistema de software para cumplir requisitos de tiempo. Cuando ocurren eventos: interrupciones, mensajes, solicitudes de usuarios u otros sistemas, o eventos del reloj que marcan el paso del tiempo; el sistema, o algún elemento del sistema, debe responder a ellos a tiempo. Caracterizar los eventos que pueden ocurrir (y cuándo pueden ocurrir) y la respuesta temporal del sistema o elemento a esos eventos es la esencia de discutir el desempeño.

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

Para un sistema basado en la web, la respuesta deseada podría expresarse como el número 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 que construir 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 sigue aumentando, otras cualidades han surgido como competidores de importación para el desempeño

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

El rendimiento a menudo está relacionado con la escalabilidad, es decir, el aumento de la capacidad de trabajo, sin dejar de funcionar bien. Técnicamente, la escalabilidad está haciendo su sistema es fácil de cambiar de una manera particular, por lo que es una especie de modificabilidad.

Además, abordamos la escalabilidad explícitamente en el Capítulo 12

8.1 Escenario general de rendimiento

Un escenario de rendimiento comienza con la llegada de un evento al sistema. Respondiendo correctamente para el evento requiere recursos (incluido el tiempo) para ser consumidos. Cuando esto está sucediendo, el sistema puede estar atendiendo simultáneamente otros eventos.

Concurrencia

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

x: = 1;
x ++;
y otro hilo que ejecuta las mismas declaraciones. Cuál es el valor

de x después de que ambos hilos hayan ejecutado esas declaraciones? Podría ser 2 o 3. Dejo que usted averigüe cómo podría ocurrir el valor 3, o ¿Debo decir que te lo entrego?

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. Varios usuarios son compatibles simultáneamente en su sistema mediante el uso de subprocesos.

La concurrencia también ocurre cada vez que su sistema se ejecuta en más de un procesador, ya sea que los procesadores estén empaquetados por separado o como procesadores multinúcleo. Además, debe considerar la concurrencia cuando algoritmos paralelos, paralelizando infraestructuras como map-reduce, o las bases de datos NoSQL son utilizadas por su sistema, o utiliza una de una variedad de algoritmos de programación concurrentes. En otras palabras, la concurrencia es una herramienta disponible para usted de muchas formas.

Concurrencia, cuando tiene varias CPU o estados de espera que pueden explotarlo, es algo bueno. Permitir que las operaciones ocurran en paralelo mejora rendimiento, porque los retrasos introducidos en un subproceso permiten que el procesador progrese en otro subproceso. Pero debido al fenómeno de entrelazado que no se acaba de describir (denominado condición de carrera), la concurrencia también debe ser administrado cuidadosamente por el arquitecto.

Como muestra el ejemplo, las condiciones de carrera pueden ocurrir cuando hay dos hilos de control y hay estado compartido. La gestión de la moneda con frecuencia se reduce a gestionar cómo se comparte el estado. Una técnica para prevenir las condiciones de carrera es usar cerraduras para hacer cumplir el acceso secuencial al estado. Otra técnica consiste en dividir el estado en función del hilo que ejecuta una parte del código. Es decir, si hay dos instancias de x en nuestro ejemplo, x no es compartido por los dos hilos y no habrá una carrera condición.

Las condiciones de carrera son uno de los tipos de errores más difíciles de descubrir; la aparición del error es esporádica y depende de diferencias (posiblemente mínimas) en el

tiempo. Una vez tuve una condición de carrera en un sistema operativo que no pudo rastrear. Pongo una prueba en el código para que la próxima vez que la carrera se produjo una condición, se desencadenó un proceso de depuración. Tomó más de un año para que el error se repita para poder determinar la causa.

No permita que las dificultades asociadas con la concurrencia lo disuadan de utilizando esta técnica tan importante. Úselo con el conocimiento de que debe identificar cuidadosamente las secciones críticas en su código y asegurarse de que las condiciones de carrera no ocurrirán en esas secciones.

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

■ Los eventos **periódicos** llegan de manera predecible a intervalos de tiempo regulares. Por ejemplo, un evento puede llegar cada 10 milisegundos. La llegada periódica de eventos es más frecuente visto en sistemas en tiempo real.

■ Llegada **estocástica** significa que los eventos llegan de acuerdo con algunos valores probabilísticos.

■ Los eventos **esporádicos** llegan de acuerdo con un patrón que no es periódico ni estocástico. Incluso estos pueden caracterizarse, sin embargo, en ciertas circunstancias. Por ejemplo, podríamos saber que como máximo 600 eventos ocurrir en un minuto, o que habrá al menos 200 milisegundos entre la llegada de dos eventos cualesquiera. (Esto podría describir un sistema en el que los eventos corresponden a las pulsaciones del teclado de un usuario humano). Son útiles caracterizaciones, aunque no sabemos cuándo ocurrirá un evento.

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

■ **Latencia**. El tiempo entre la llegada del estímulo y el sistema responde a ella.

■ **Plazos de tramitación**. (deadline) 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** (throughput), generalmente expresado como el número de transacciones que sistema puede procesar en una unidad de tiempo.

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

■ El número de **eventos no procesados** porque el sistema estaba demasiado ocupado para responder.

A partir de estas consideraciones podemos ahora describir las porciones individuales de un

escenario general de rendimiento:

■ **Fuente de estímulo**. Los estímulos llegan desde el exterior (posiblemente múltiples) o fuentes internas.

■ **Estímulo**. Los estímulos son las llegadas de eventos. 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, de modo normal a modo de sobrecarga).

■ **Medida de respuesta**. Las medidas de respuesta son el tiempo que lleva procesar los eventos que llegan (latencia o una 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 se pueden procesar (tasa de fallos).

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

La figura 8.1 ofrece 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.

Tabla 8.1 Escenario general de rendimiento

Porción de valores	**posibles del escenario**
Fuente	Interna o externa al sistema
Estímulo	Llegada de un evento periódico, esporádico o estocástico
Artifacto	System o uno o más componentes del sistema
Entorno	Modo operativo: normal, emergencia, carga máxima, sobrecarga
Respuesta	Eventos del proceso de respuesta, cambio de nivel de servicio
Medida de respuesta	Latencia, fecha límite, rendimiento, fluctuación, tasa de errores

8.2 – Tactica para Perfomance

El objetivo de las tácticas de actuación es generar una respuesta a la llegada de un evento en el sistema dentro de alguna restricción basada en el tiempo. El evento puede ser único o un stream y es el disparador para realizar el cálculo. Las tácticas de desempeño controlan el tiempo dentro del cual se genera una respuesta, como se ilustra en la Figura 8.2.

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

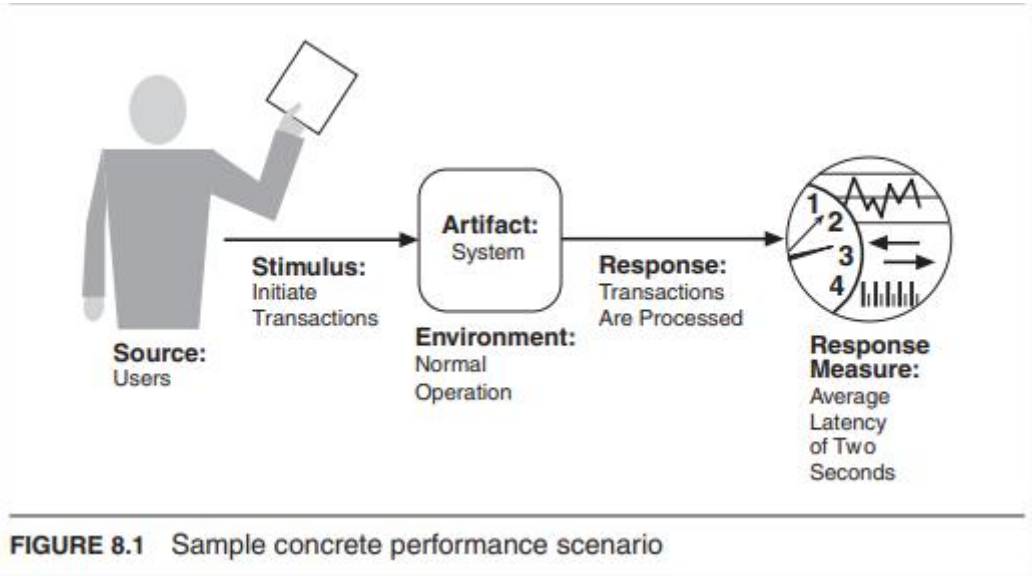

FIGURE 8.1 Sample concrete performance scenario

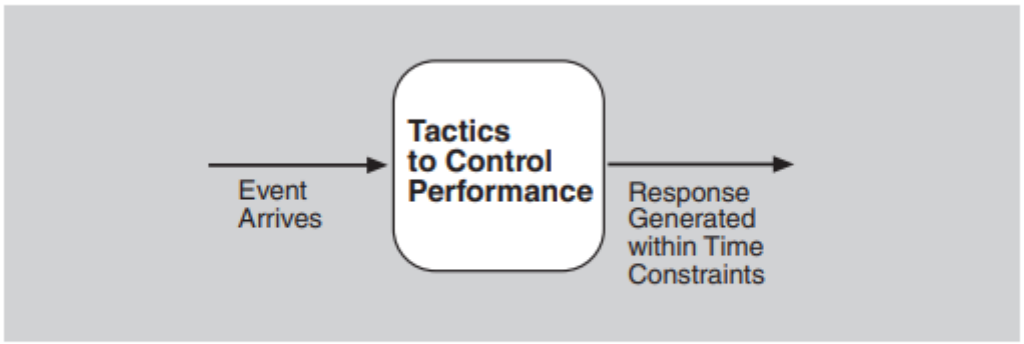

FIGURE 8.2 The goal of performance tactics

■ **Tiempo de procesamiento.** El procesamiento consume recursos, lo que lleva tiempo. Eventos son manejados por la ejecución de uno o más componentes, cuyo tiempo gastado es un recurso. Los recursos de hardware incluyen CPU, almacenes de datos, ancho de banda de comunicación de red y memoria. Recursos de software incluyen entidades definidas por el sistema bajo diseño. Por ejemplo, búferes deben ser gestionados y acceder a secciones críticas y debe hacerse secuencial.

Por ejemplo, suponga que un componente genera un mensaje. Eso puede colocarse en la red, después de lo cual llega a otro componente. Luego se coloca en un búfer; transformado de alguna manera; procesado según algún algoritmo; transformado para la producción; colocado en una salida buffer; 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 se acerca su utilización su capacidad, es decir, a medida que se saturan. Por ejemplo, como CPU se carga más, el rendimiento generalmente se degrada de manera bastante constante.

Por otro lado, cuando comienza a quedarse sin memoria, en algún momento el intercambio de páginas se vuelve abrumador y el rendimiento se bloquea repentinamente**.**

■ **Tiempo bloqueado.** Un cálculo puede bloquearse debido a la contención de algunos recurso necesario, porque el recurso no está disponible o porque el cálculo depende del resultado de otros cálculos que aún no están disponibles:

- **Contención de recursos.** Muchos recursos solo pueden ser utilizados por una cliente a la vez. Esto significa que otros clientes deben esperar el acceso a esos recursos. La figura 8.2 muestra los eventos que llegan al sistema. Estas los eventos pueden estar en un solo flujo o en múltiples flujos. Varias corrientes compitiendo por el mismo recurso o diferentes eventos en la misma transmisión compitiendo porque el mismo recurso contribuye a la latencia. La mayor contienda por un recurso, mayor probabilidad de que se introduzca latencia.
- **Disponibilidad de recursos.** Incluso en ausencia de contención, el cálculo no puede continuar si un recurso no está disponible. La indisponibilidad puede ser causada porque el recurso está fuera de línea o por falla del componente o por alguna otra razon. En cualquier caso, debe identificar los lugares donde la falta de disponibilidad de recursos podría causar una contribución significativa a la latencia general. Algunos de nuestras tácticas están destinados a hacer frente a esta situación.
- **Dependencia de otros cálculos**. Es posible que un cálculo tenga 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 responder, el tiempo puede ser significativo si el componente llamado está en el otro extremo de una red (a diferencia de coubicado en el mismo procesador).

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

- **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 dar servicio a los eventos.

- **Administrar recursos**. Esta táctica opera en el lado de la respuesta para hacer que los recursos disponibles trabajen de manera más efectiva en el manejo de las demandas que se les plantean.

Controlar la demanda de recursos

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

■ **Gestionar la frecuencia de muestreo.** Si es posible reducir la frecuencia de muestreo en el que se captura un flujo de datos ambientales, entonces la demanda puede ser reducido, por lo general con cierta pérdida de fidelidad concomitante. Esto es común en sistemas de procesamiento de señales donde, por ejemplo, se pueden utilizar diferentes códecs elegido con diferentes velocidades de muestreo y formatos de datos. Esta elección de diseño está hecha para mantener niveles predecibles de latencia; debes decidir si tener una menor fidelidad, pero un flujo de datos consistente es preferible a perder paquetes de datos.

■ **Limite la respuesta a eventos**. Cuando llegan eventos discretos al sistema (o elemento) demasiado rápido para ser procesados, entonces los eventos deben ponerse en cola hasta que puedan estar procesado. Debido a que estos eventos son discretos, generalmente no es deseable para "reducir la resolución". En tal caso, puede optar por procesar eventos solo hasta una tasa máxima establecida, lo que garantiza un procesamiento más predecible cuando los eventos se procesan realmente. Esta táctica podría activarse por un tamaño de cola o una medida de utilización del procesador que supere alguna advertencia nivel. Si adopta esta táctica y es inaceptable perder algún evento, entonces debe asegurarse de que sus colas sean lo suficientemente grandes para manejar el peor de los casos.

Si, por el contrario, elige eliminar eventos, debe elegir una política para manejar esta situación: ¿registra los eventos eliminados o simplemente ¿ingnóralos? ¿Notifica a otros sistemas, usuarios o administradores?

■ **Priorizar eventos**. Si no todos los eventos son igualmente importantes, puede imponer un esquema de prioridad que clasifica los eventos de acuerdo con la importancia del servicio ellos. 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 a un sistema que atiende todos los eventos todo el tiempo. Por ejemplo, un sistema de gestión de edificios puede generar una variedad de alarmas. Alarmas que amenazan la vida como una alarma de incendio debe tener mayor prioridad que la información alarmas como una habitación demasiado fría.

■ **Reducir los gastos generales (overhead).** El uso de intermediarios (tan importante para la modificabilidad, como vimos en el Capítulo 7) aumenta los recursos consumidos en el procesamiento un flujo de eventos, por lo que eliminarlos mejora la latencia. Ésta es una compensación clásica entre modificabilidad y rendimiento. Separación de preocupaciones, otro eje de la modificabilidad, también puede aumentar la sobrecarga de procesamiento necesaria para dar servicio a un evento si lleva a que un evento sea atendido por una cadena de componentes en lugar de un solo componente. El cambio de contexto y los costos de

comunicación entre componentes se acumulan, especialmente cuando los componentes están en diferentes nodos de una red. Una estrategia para reducir la sobrecarga computacional es coubicar los recursos. La coubicación puede significar alojamiento componentes que cooperan en el mismo procesador para evitar el retraso de tiempo de red de comunicacion; puede significar poner los recursos en el mismo componente de software en tiempo de ejecución para evitar incluso el gasto de una llamada de subrutina.

Un caso especial de reducir la sobrecarga computacional es realizar una limpieza de recursos que se han vuelto ineficientes. Por ejemplo, tablas hash y los mapas de memoria virtual pueden requerir un nuevo cálculo y reinicialización.

Otra estrategia común es ejecutar servidores de un solo subproceso (para simplificar y evitar la contención) y dividir la carga de trabajo entre ellos.

■ **Tiempos de ejecución acotados**. Ponga un límite a la cantidad de tiempo de ejecución que se utiliza para responder a un evento. Para algoritmos iterativos dependientes de datos, limitar el número de iteraciones es un método para limitar los tiempos de ejecución. El precio es generalmente un cálculo menos preciso. Si adopta esta táctica, necesitará para evaluar su efecto en la precisión y ver si el resultado es "suficientemente bueno". Esta La táctica de gestión de recursos se combina con frecuencia con la gestión de muestreo. táctica de tasa.

■ **Incrementar la eficiencia de los recursos.** Mejorando los algoritmos utilizados en áreas críticas disminuirá la latencia.

Administrar recursos

Incluso si la demanda de recursos no es controlable, la gestión de estos recursos puede serlo. A veces, un recurso puede cambiarse por otro. Por ejemplo, los datos intermedios pueden guardarse en un caché o pueden regenerarse dependiendo de disponibilidad de recursos de tiempo y espacio. Esta táctica se suele aplicar al procesador, pero también es eficaz cuando se aplica a otros recursos como un disco. **Aquí están algunas tácticas de gestión de recursos:**

■ **Incrementar los recursos.** Procesadores más rápidos, procesadores adicionales, adicionales la memoria y las redes más rápidas tienen el potencial de reducir la latencia.

El costo suele ser 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 barata de obtener una mejora inmediata.

■ **Introducir la concurrencia.** Si las solicitudes se pueden procesar en paralelo, el bloqueado el tiempo se puede reducir. La concurrencia se puede introducir procesando diferentes flujos de eventos en diferentes subprocesos o creando subprocesos adicionales para procesar diferentes conjuntos de actividades. Una vez que se ha introducido la concurrencia , se pueden utilizar políticas de programación para lograr los objetivos que considere deseables. Diferentes políticas de programación pueden maximizar la equidad (todas las solicitudes igual tiempo), rendimiento (tiempo más corto para terminar primero) u otros objetivos. (Ver la barra lateral.)

■ **Mantenga múltiples copias de los cálculos**. Varios servidores en un patrón cliente-servidor son réplicas de la computación. El propósito de las réplicas es reducir la contención que ocurriría si todos los cálculos tuvieran lugar en un solo servidor. Un balanceador de carga es una pieza de software que asigna nuevo trabajo a uno de los servidores duplicados

disponibles; Los criterios de asignación varían, pero pueden ser tan simple como round-robin o asignar la siguiente solicitud al servidor menos ocupado.

■ **Mantenga múltiples copias de datos.** El almacenamiento en caché es una táctica que implica mantener copias de datos (posiblemente uno un subconjunto del otro) en almacenamiento con diferentes velocidades de acceso. Las diferentes velocidades de acceso pueden ser inherentes (memoria versus almacenamiento secundario) o puede deberse a la necesidad de comunicación de red. La replicación de datos implica mantener copias separadas de los datos para reducir la contención de múltiples accesos simultáneos. Porque los datos son almacenado en caché o replicado suele ser una copia de los datos existentes, manteniendo las copias coherente y sincronizado se convierte en una responsabilidad que el sistema debe asumir. Otra responsabilidad es elegir los datos que se almacenarán en caché. Algunos los cachés funcionan simplemente manteniendo copias de lo que se solicitó recientemente, pero también es posible predecir las solicitudes futuras de los usuarios en función de patrones de comportamiento, y comenzar los cálculos o búsquedas previas necesarias para cumplir con esas solicitudes antes de que el usuario las haya realizado.

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

■ **Programar recursos.** Siempre que haya contienda por un recurso, el recurso debe ser programado. Los procesadores están programados, los búferes programados y las redes están programadas. Tu 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 desempeño se resumen en la Figura 8.3.

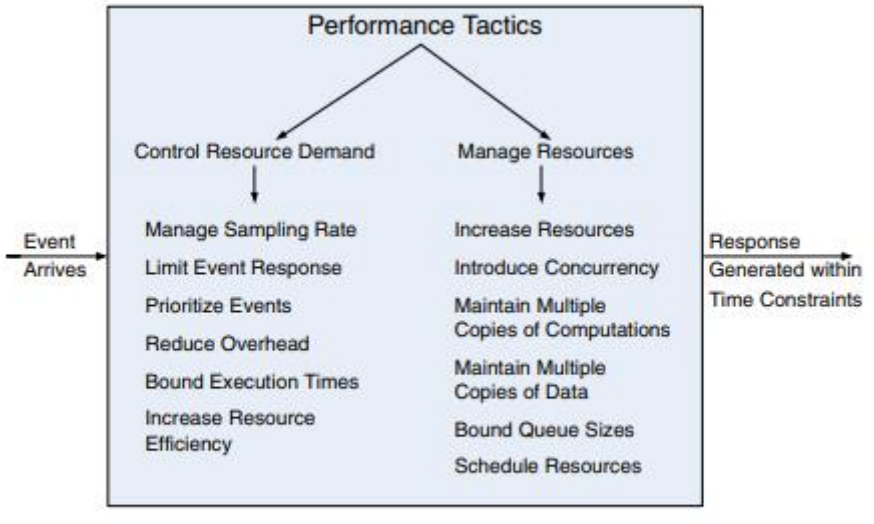

FIGURE 8.3 Performance tactics

Políticas de programación

Una política de programación tiene conceptualmente dos partes: una asignación de prioridad y despacho. 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 que compiten para la programación incluyen el uso óptimo de recursos, solicitud importancia, minimizando la cantidad de recursos utilizados, minimizando la latencia, maximizar el rendimiento, prevenir el hambre para garantizar la equidad, y así adelante. Debe ser consciente de estos criterios posiblemente contradictorios y efecto que tiene la táctica elegida en su encuentro.

Un flujo de eventos de alta prioridad se puede enviar solo si el recurso para que se le asigna está disponible. A veces, esto depende de adelantarse al usuario actual del recurso. Las posibles opciones de preferencia son de la siguiente manera: puede ocurrir en cualquier momento, puede ocurrir solo en puntos de preferencia específicos, y los procesos en ejecución no se pueden adelantar. Algunas políticas de programación son estas:

■ **Primero en entrar / primero en salir**. Las colas **FIFO** tratan todas las solicitudes de recursos como iguales y satisfacerlos a su vez. Una posibilidad con una cola FIFO es que la solicitud se quedará atascada detrás de otra que tarda mucho en generar una respuesta. Siempre que 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. (Fixed-priority scheduling)** La programación de prioridad fija asigna cada fuente de recurso solicita una prioridad particular y asigna los recursos en ese orden de prioridad. Esta estrategia asegura un mejor servicio para una mayor prioridad peticiones. Pero admite la posibilidad de una prioridad menor, pero importante, que se lleve un tiempo arbitrariamente largo para ser reparado, porque está atascado detrás de una serie de solicitudes de mayor prioridad. Tres estrategias de priorizaciones comunes son estas:

- **Importancia semántica**. A cada flujo se le asigna una prioridad estáticamente según algún dominio característico de la tarea que lo genera.
- **Plazo monótono (Deadline monotinic)**. La fecha límite monótona es una asignación de prioridad estática que asigna mayor prioridad a las transmisiones con plazos más cortos. Esta política de programación se utiliza cuando las transmisiones de se programarán diferentes prioridades con plazos en tiempo real.
- **Califique monótona (Rate monotinic)**. Tasa monótona es una asignación de prioridad estática para las transmisiones periódicas que asigna mayor prioridad a las transmisiones con períodos más cortos. Esta política de programación es un caso especial de fecha límite monótono, pero es más conocido y más probable que sea apoyado por el sistema operativo.

■ **Programación de prioridad dinámica (Dynamic priority scheduling)**. 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 de round-robin es un ejecutivo cíclico, donde las posibilidades de asignación son a intervalos de tiempo fijos.

- **Primera fecha límite primero (Earliest-deadline-first)**. Fecha límite más temprana primero. Fecha límite más temprana primero asigna prioridades basadas en las solicitudes pendientes con las primeras fecha límite.
- **Menos holgura primero. (Least-slack-first)** Esta estrategia asigna la máxima prioridad 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 interrumpibles (es decir, es posible suspender el procesamiento de una tarea para dar servicio a una tarea cuya fecha límite se acerca), tanto la estrategia de programación de fecha límite más temprana como la de menor holgura son óptimas. Es decir, si el conjunto de procesos puede programarse de modo que se cumplan todos los plazos, entonces estas estrategias serán capaces de programar ese conjunto con éxito.

■ **Programación estática.** Un horario ejecutivo cíclico es una estrategia de programación donde los puntos de preferencia y la secuencia de asignación a los recursos se determinan fuera de línea. La sobrecarga de tiempo de ejecución de un planificador es así obviada.

Tácticas de rendimiento en la carretera

Las tácticas son principios de diseño genéricos. Para ejercitar este punto, piense en el diseño de los sistemas de carreteras y autopistas donde vive. Los ingenieros emplean una serie de "**trucos" de diseño para optimizar el rendimiento de estos sistemas complejos,** donde el rendimiento tiene una serie de medidas, como el rendimiento (cuántos coches por hora llegan de los suburbios al estadio de fútbol), latencia promedio de casos (cuánto tiempo se tarda, en promedio, en llegar de su casa al centro) y latencia en el peor de los casos (¿Cuánto tiempo se necesita un vehículo de emergencia para llevarlo al hospital? ¿Qué son estos trucos? Nada menos que nuestros buenos y viejos amigos, tácticas.

Consideremos algunos **ejemplos**:

■ Administrar la tasa de eventos. Las luces en las rampas de entrada de la autopista permiten que los autos entren autopista solo a intervalos establecidos, y los automóviles deben esperar (hacer cola) en la rampa parasu turno.

■ Priorizar eventos. Ambulancias y policías, con sus luces y sirenas van, 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.

■ Mantenga varias copias. Agregue carriles de tráfico a las carreteras existentes o construya rutas paralelas. Además, existen algunos trucos que los usuarios del sistema pueden emplear:

■ Incrementar los recursos. Compra un Ferrari, por ejemplo. Todas las demás cosas son igual, el automóvil más rápido con un conductor competente en una carretera abierta obtendrá llegar a su destino más rápidamente.

■ Incrementar 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 en frente a usted, o puede cargar más personas en el mismo vehículo (es decir, coche compartido).

¿Cuál es el punto de esta discusión? Parafraseando a Gertrude Stein: rendimiento es rendimiento es rendimiento. Los ingenieros han estado analizando y optimizando sistemas durante siglos, tratando de mejorar su rendimiento, y han estado empleando las mismas estrategias de diseño para hacerlo. Entonces debe sentirse un poco reconfortado al saber que cuando intenta mejorar la rendimiento de su sistema basado en computadora, está aplicando tácticas que han sido completamente "probados en carretera".

—RK