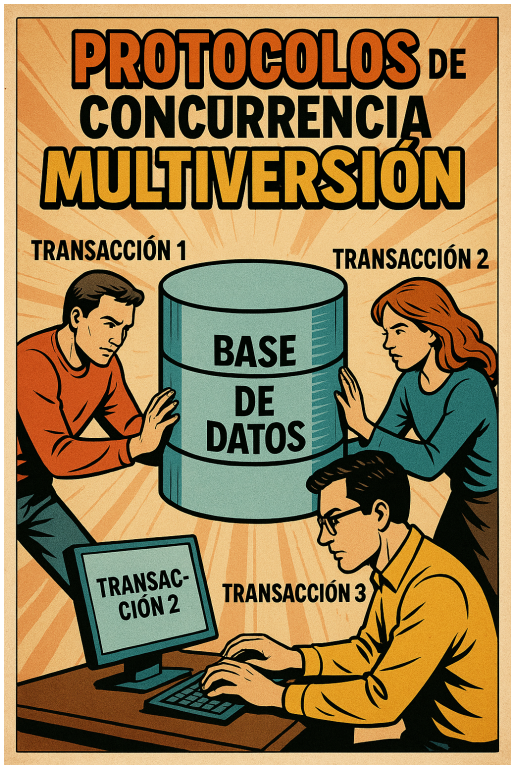
Protocolos

- MVTO multiversion timestamp ordering
- MV2PL multiversion two-phase locking
  - 2V2PL two version two-phase locking
- ROMV Read Only Multiversion



Multiversion Timestamp Ordering

Cada versión de un elemento de datos lleva un timestamp $ts(t_i)$ de la transacción t_i que fue la que creo la versión.

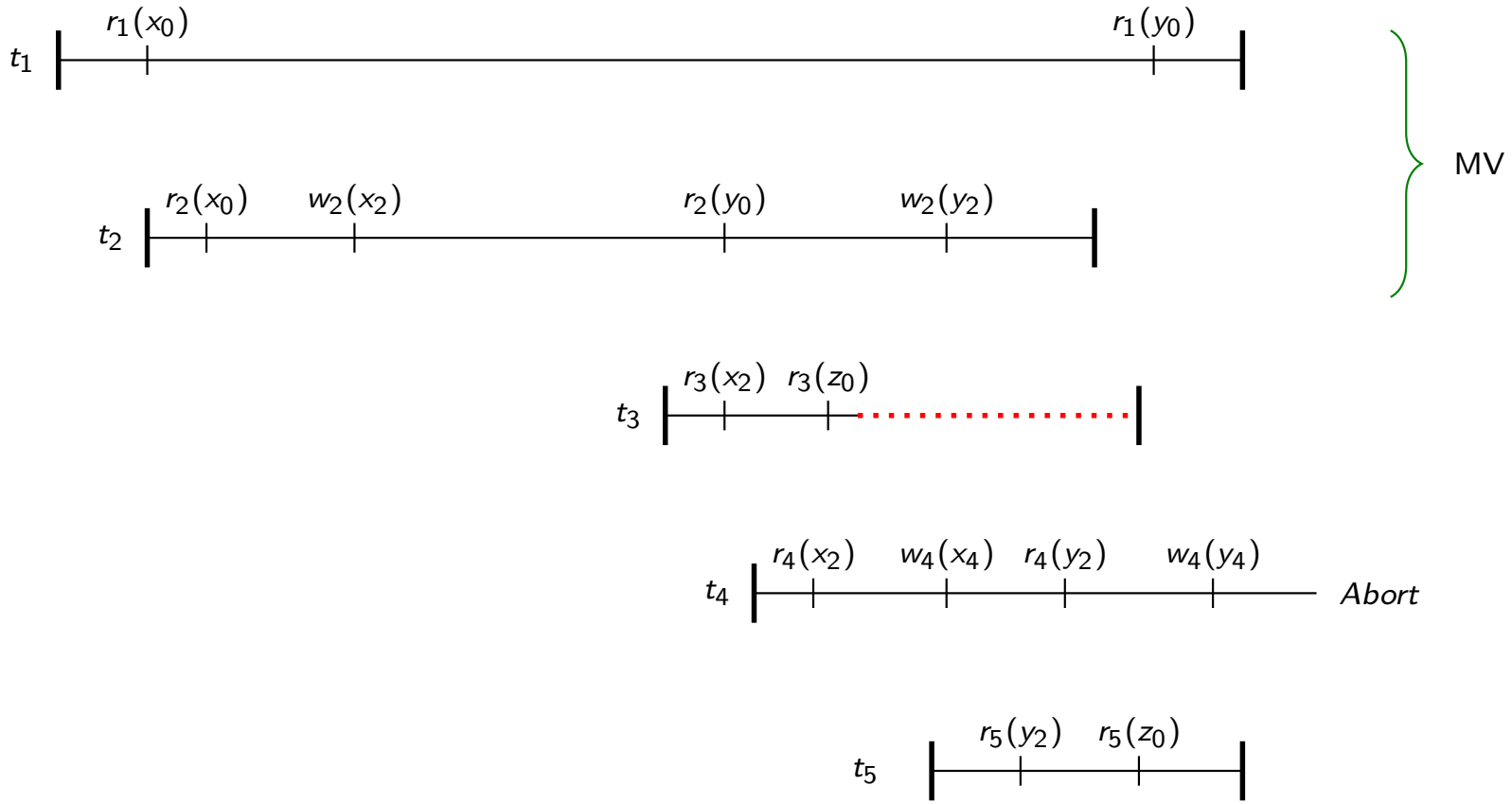
1. Una operación $r_i(x)$ se transforma en una operación multiversión $r_i(w_k)$ donde w_k es la versión de x que tiene el timestamp mas grande menor o igual que $ts(t_i)$ y que fue escrita por $t_k, k \neq i$
2. Una operación $w_i(x)$ se procesa de la siguiente manera
   - Si una operación $r_j(x_k)$ tal que $ts(t_k) < ts(t_i) < ts(t_j)$ ya existe en el schedule entonces $w_i(x)$ es rechada y t_i es abortada. Se produce un *write too late*.
   - en otro caso $w_i(x)$ se transforma en $w_i(x_i)$ y es ejecutada.
3. Un commit c_i se retrasa hasta que los commit c_j de todas las transacciones t_j que han escrito nuevas versiones de los elementos de datos leídos por t_i hayan sido ejecutados.

Ejemplo MVTO

$r_1(x); r_2(x); w_2(x); r_3(x); r_2(y); r_4(x); r_3(Z); w_4(X); w_2(y); r_5(y); r_4(y); r_5(z); r_1(y); w_4(y)$

Ejemplo MVTO

$r_1(x); r_2(x); w_2(x); r_3(x); r_2(y); r_4(x); r_3(Z); w_4(X); w_2(y); r_5(y); r_4(y); r_5(z); r_1(y); w_4(y)$



Multiversion two-phase Locking

MV2PL

El MV2PL es un protocolo basado el locking usando *strong strict two-phase locking* o **2PL riguroso**.

1. *versiones commiteadas*, que han sido escritas por transacciones que ya están commiteadas,
2. *versión actual* es la versión commiteada de ese elemento de datos escrita por la última transacción commiteada;
3. *versiones no commiteadas*, que son todas las versiones restantes (creadas por transacciones que todavía están activas).

Multiversion two-phase Locking

1. Si el paso no es el final dentro de una transacción:
   1. Una lectura $r(x)$ se ejecuta de inmediato, asignándole la versión actual del elemento de datos solicitado, es decir, la versión commiteada más recientemente (pero no cualquier otra, previamente commiteada), o asignándole una versión no commiteada de x
   2. Un escritura $w(x)$ se ejecuta solamente cuando la transacción que ha escrito x por última vez finalizo, es decir no hay otra versión no commiteada de x. O sea se libero el lock de escritura sobre x
2. Si es el paso final de la transacción t_i, esta se retrasa hasta que commitean las siguientes transacciones
   1. todas aquellas transacciones t_j que hayan leído la versión actual de un elemento de datos escrito por t_i
   2. Todas aquellas t_j de las que t_i ha leído algún elemento.

Ejemplo MV2PL

$$s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_2(x)c_2w_1(y)c_1$$

1. $r_1(x)$ is assigned to x_0 and is executed: $r_1(x_0)$
2. $w_1(x)$ is executed since no other transaction is still active: $w_1(x_1)$
3. let $r_2(x)$ be assigned to x_1 and executed: $r_2(x_1)$
4. $w_2(y)$ is executed: $w_2(y_2)$
5. let $r_1(y)$ be assigned to y_0 and executed: $r_1(y_0)$
6. if $w_2(x)$ were *not* the final step of t_2, it would be delayed since t_1 is still active and has written x_1. However, as it *is* the final step of t_2, the final-step rules need to be applied. It turns out that t_2 nonetheless has to wait for the following reason:
   (a) t_1 has read the current version of data item y (y_0), and t_2 overwrites this version,
   (b) t_2 has read x_1 from t_1.
7. $w_1(y)$, the final step of t_1, is executed since
   (a) t_2 has *not* read a current version of a data item written by t_1 (current versions are x_0, y_0),
   (b) t_1 has not read a version written by t_2. $w_1(y_1)$
8. finally, $w_2(x)$ can be executed: $w_2(x_2)$

2V2PL protocol

El protocolo 2V2PL (bloqueo de dos versiones) mantiene como máximo dos versiones de cualquier elemento de datos en cada momento.

- Supongamos que t_i escribe el elemento de datos x, pero aún no está *comiteado*, las dos versiones de x son su imagen anterior y su imagen posterior.
- Tan pronto como t_i se comitea, la imagen anterior puede ser eliminada ya que la nueva versión de x ahora es estable, y las versiones antiguas ya no son necesarias ni se mantienen.

En 2V2PL las operaciones de lectura están restringidas a leer solo las versiones actuales, es decir, la última versión comiteada

2V2PL protocol - Locks

- *rl* **read lock**: que se establece antes de una operación de lectura $r(x)$ respecto de la versión actual de x
- *wl* **write lock**; que se establece antes de una operación de escritura $w(x)$ para escribir una versión no commiteada de x.
- *cl* **commit o certify lock**:se establece un $cl(x)$ antes de la ejecución del paso final de una transacción en cada elemento de datos x que esta transacción **ha escrito**.

2V2PL

Las operaciones de unlock deben obedecer al protocolo **2PL**. La matriz de compatibilidad es la siguiente

	rl(x)	*wl(x)*	*cl(x)*
rl(x)	+	+	−
wl(x)	+	−	−
cl(x)	−	−	−

Ejercicio

Suponga el siguiente schedule que se quiere ejecutar sobre un motor que soporta **2V2PL**.

$$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$$

2V2PL

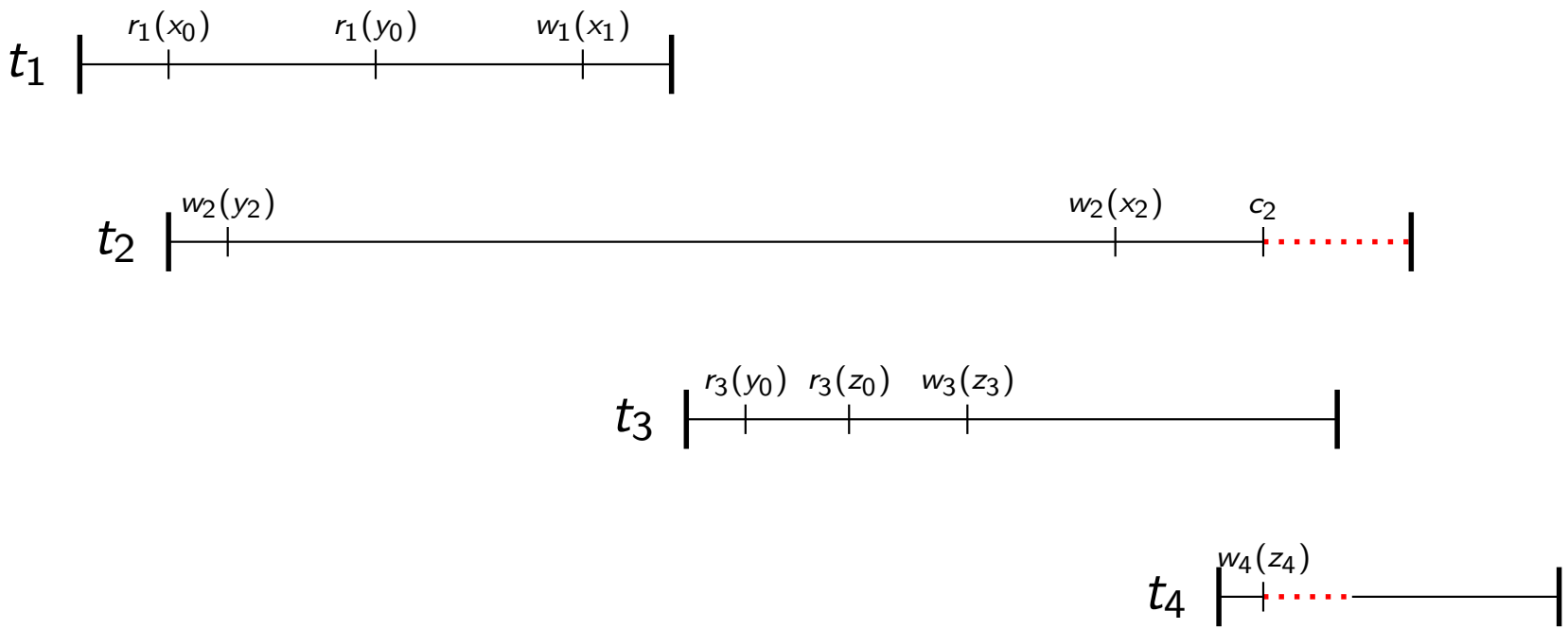
Indicar que ocurre en cada paso y en que lugar las operaciones deben ser demoradas. Escriba el *schedule* final indicando *locks y unlocks*.

Solución

$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$

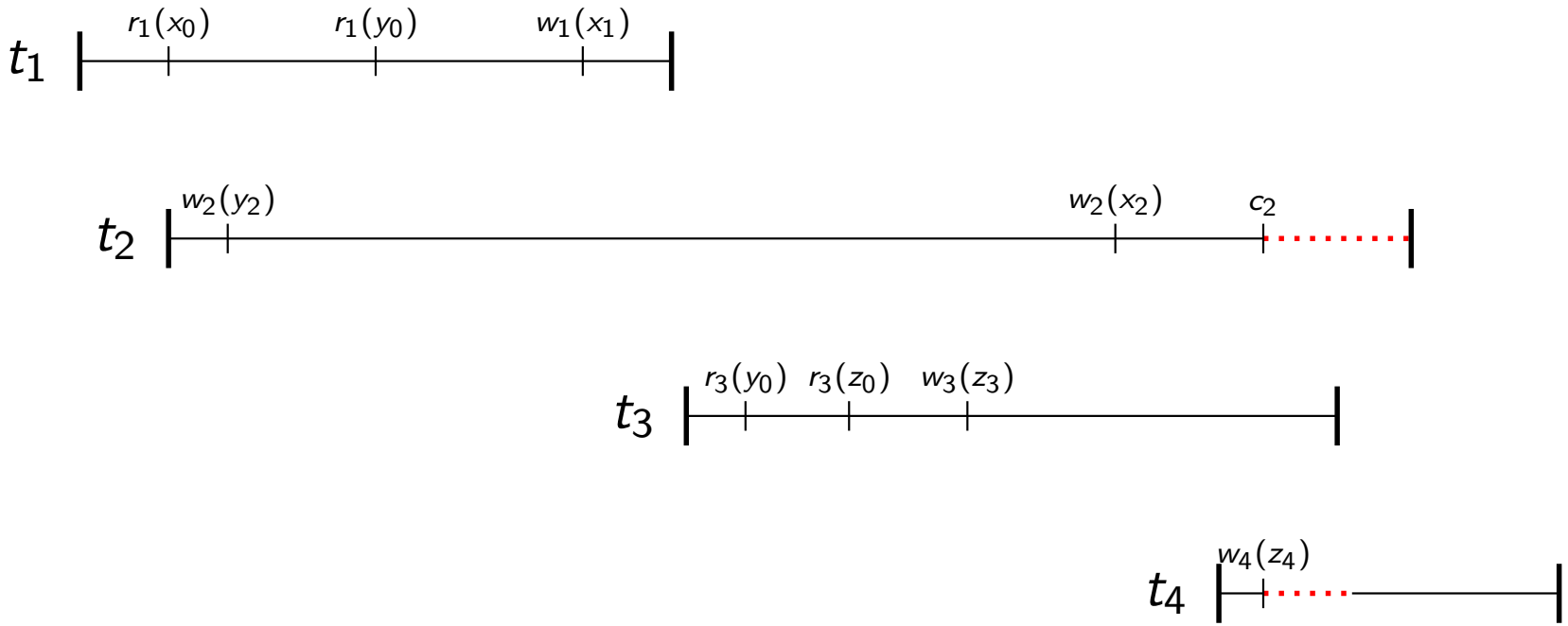
Solución

$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$



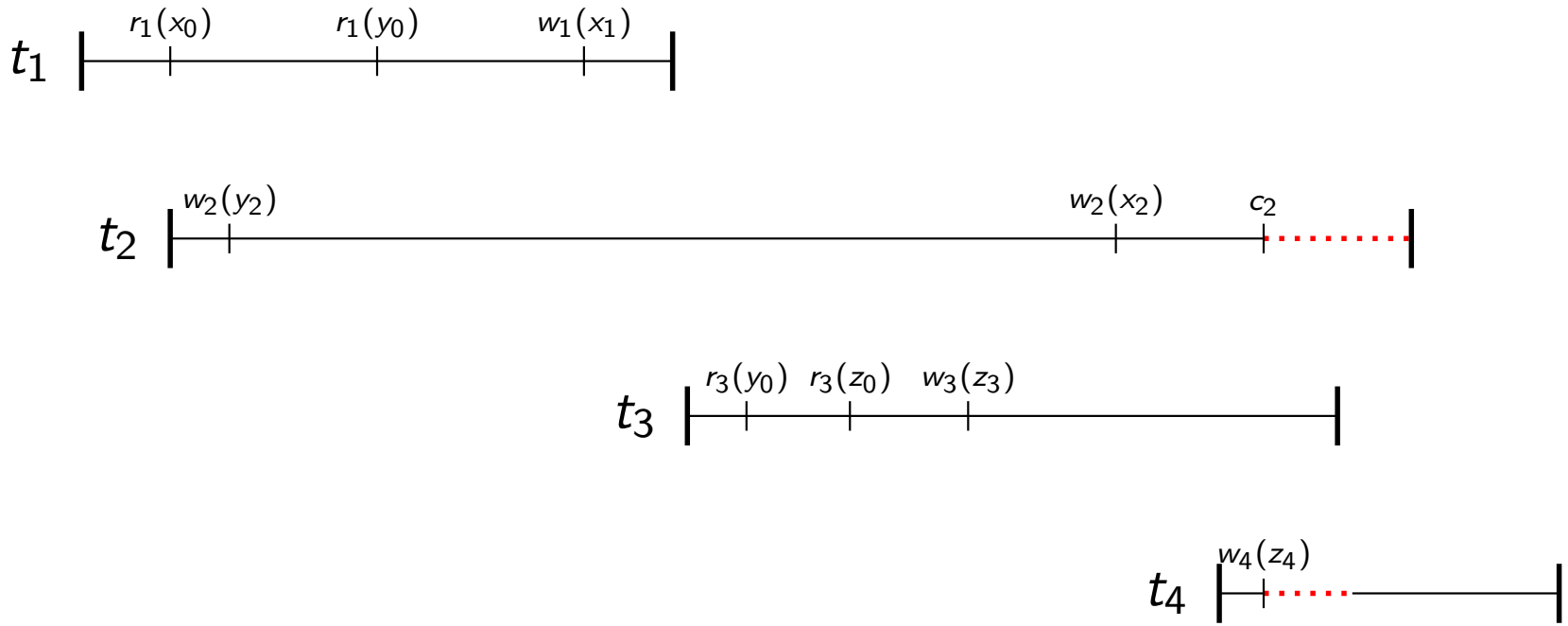
Solución

$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$

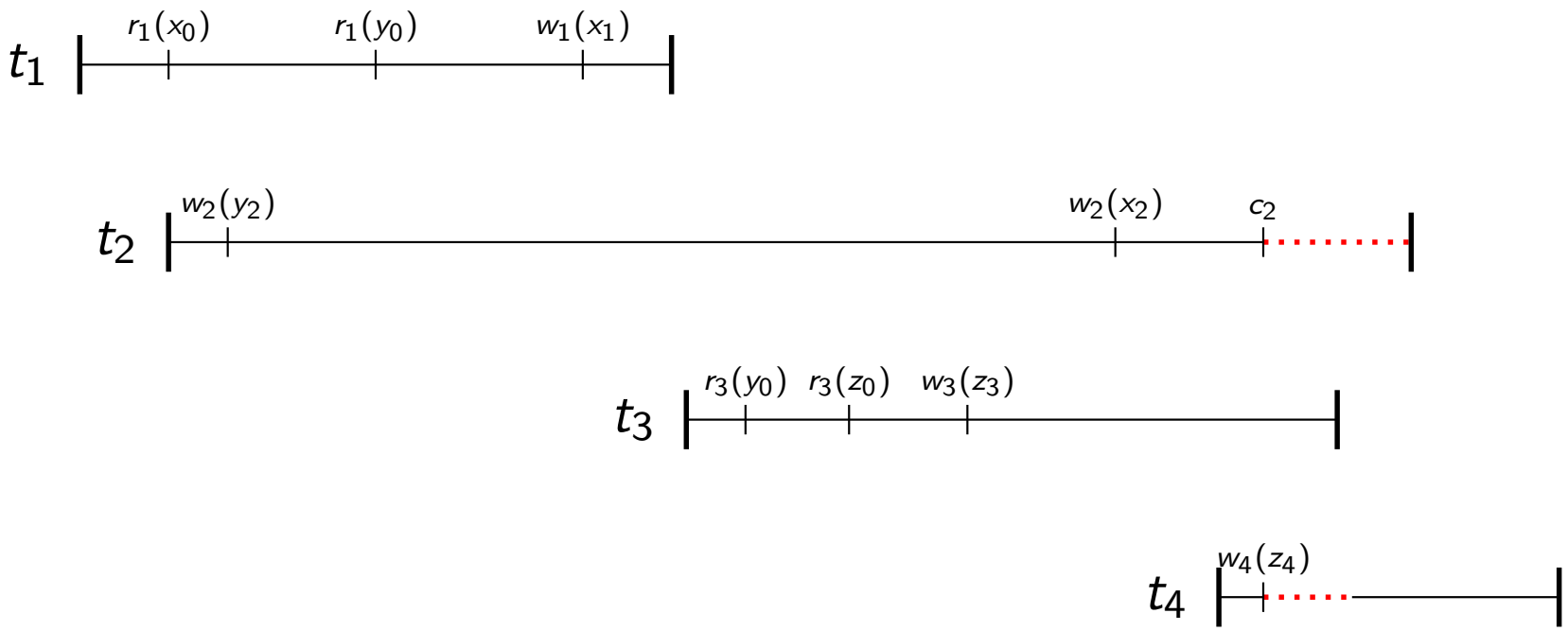


Solución

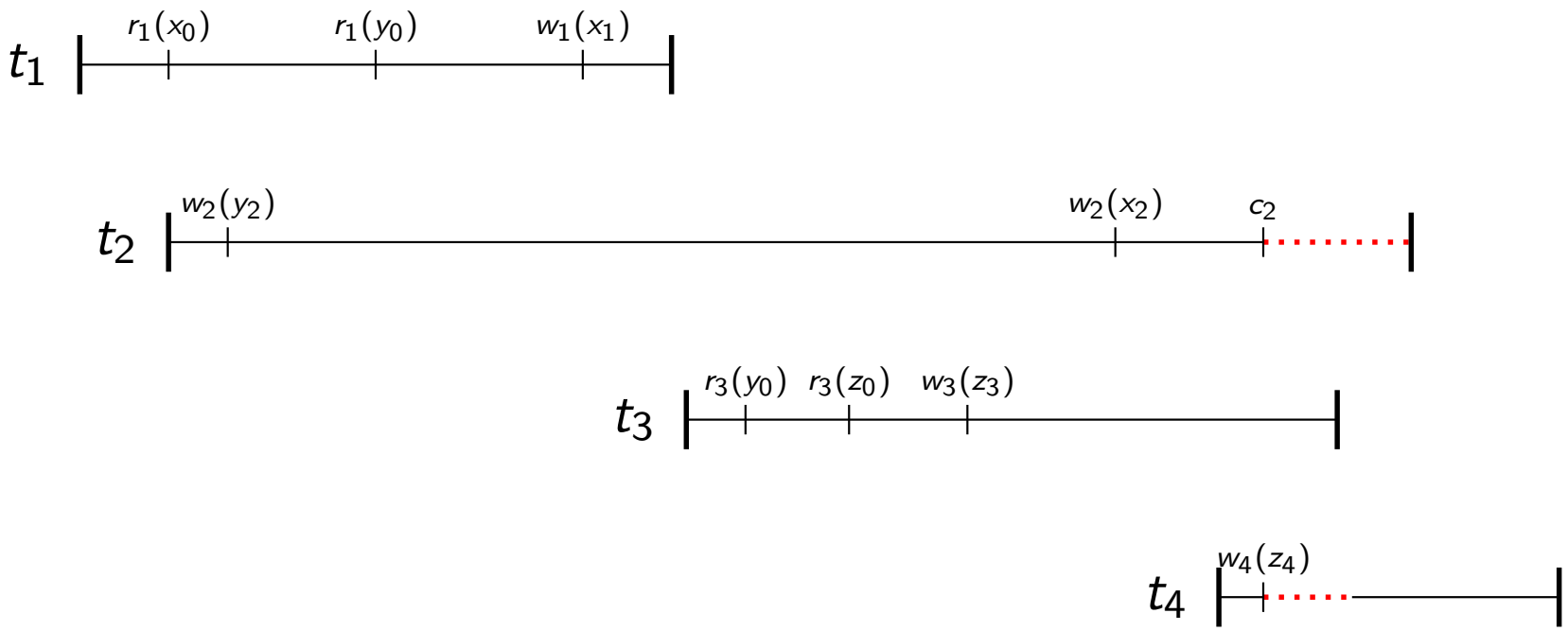
$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$



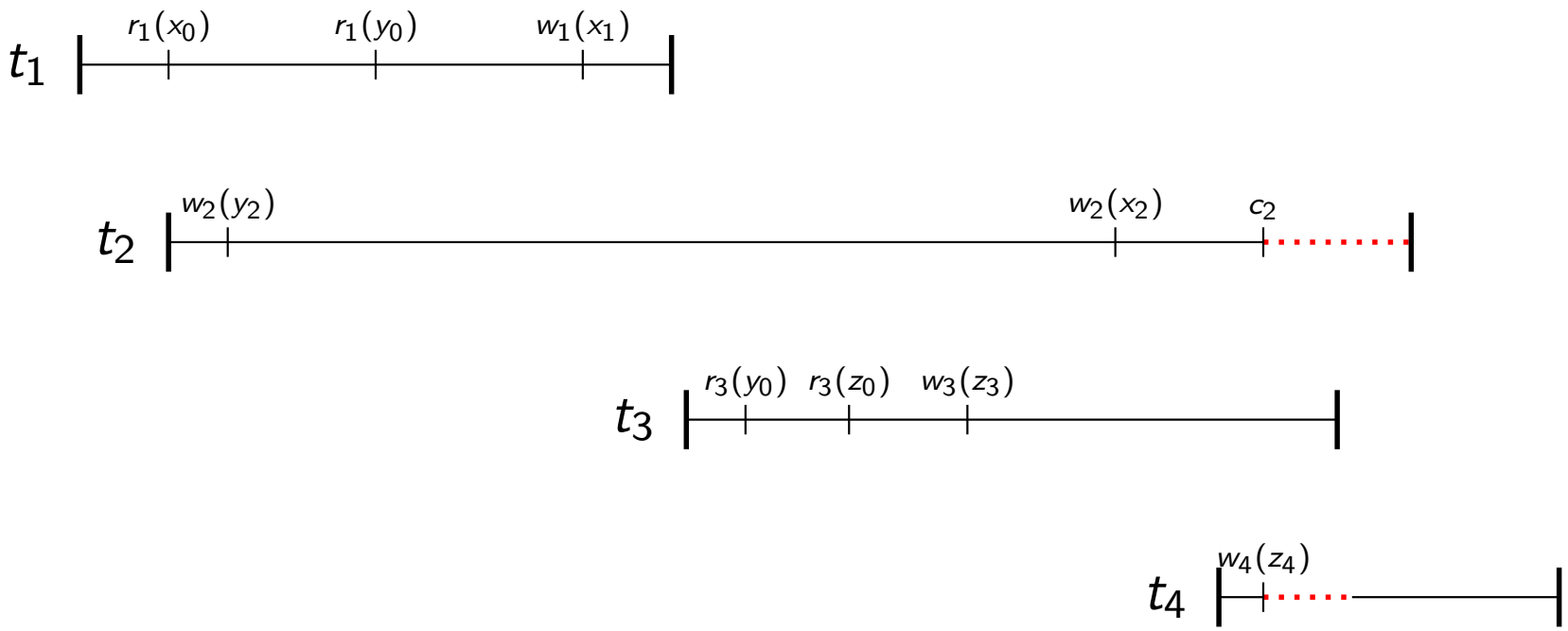
$rl_1(x); r_1(x_0); wl_2(y); w_2(y_2); rl_1(y); r_1(y_0); wl_1(x); w_1(x_1); cl_1(x), ul_1;$
$c_1; rl_3(y); r_3(y_0); rl_3(z); r_3(z_0); wl_3(z); w_3(z_3); wl_2(x); w_2(x_2);$
$cl_2(x); cl_3(z); ul_3; c_3; cl_2(y); ul_2; c_2; w_4(z_4); cl_4(z); ul_4; c_4$



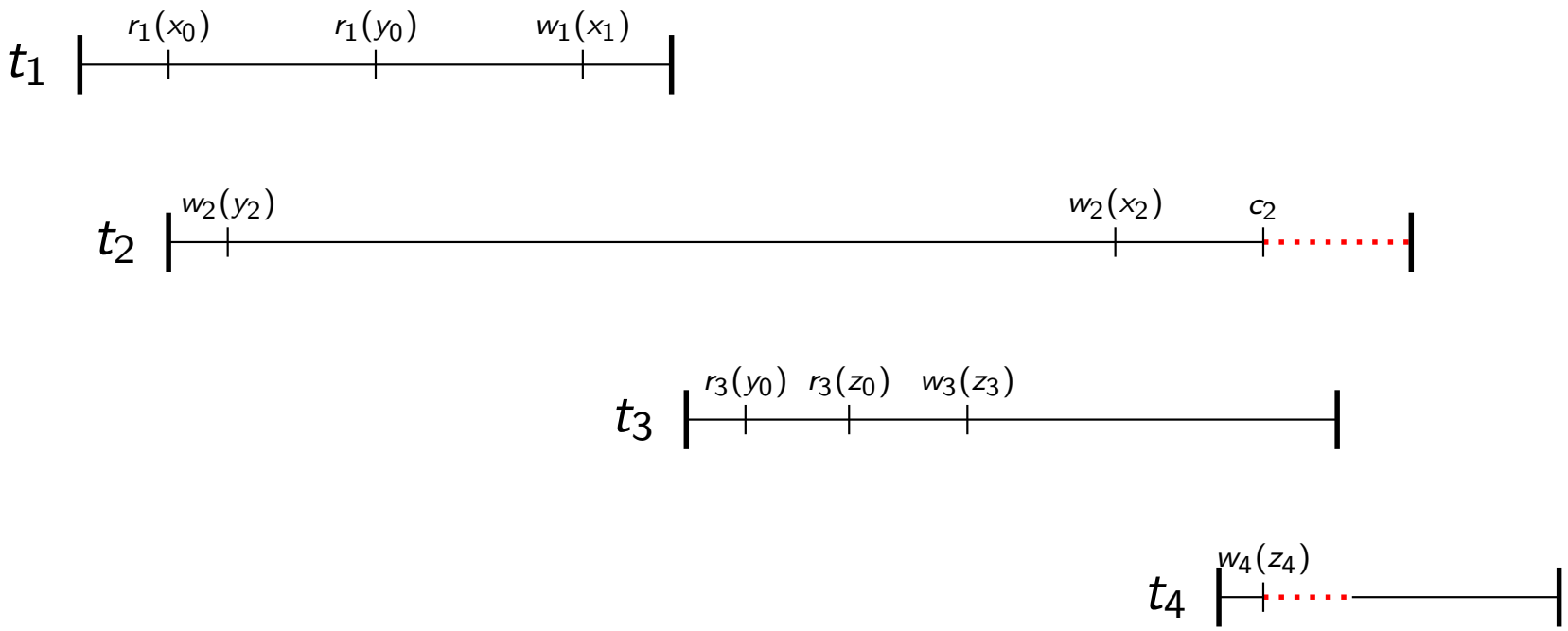
X	Y	Z
x_0	y_0	z_0



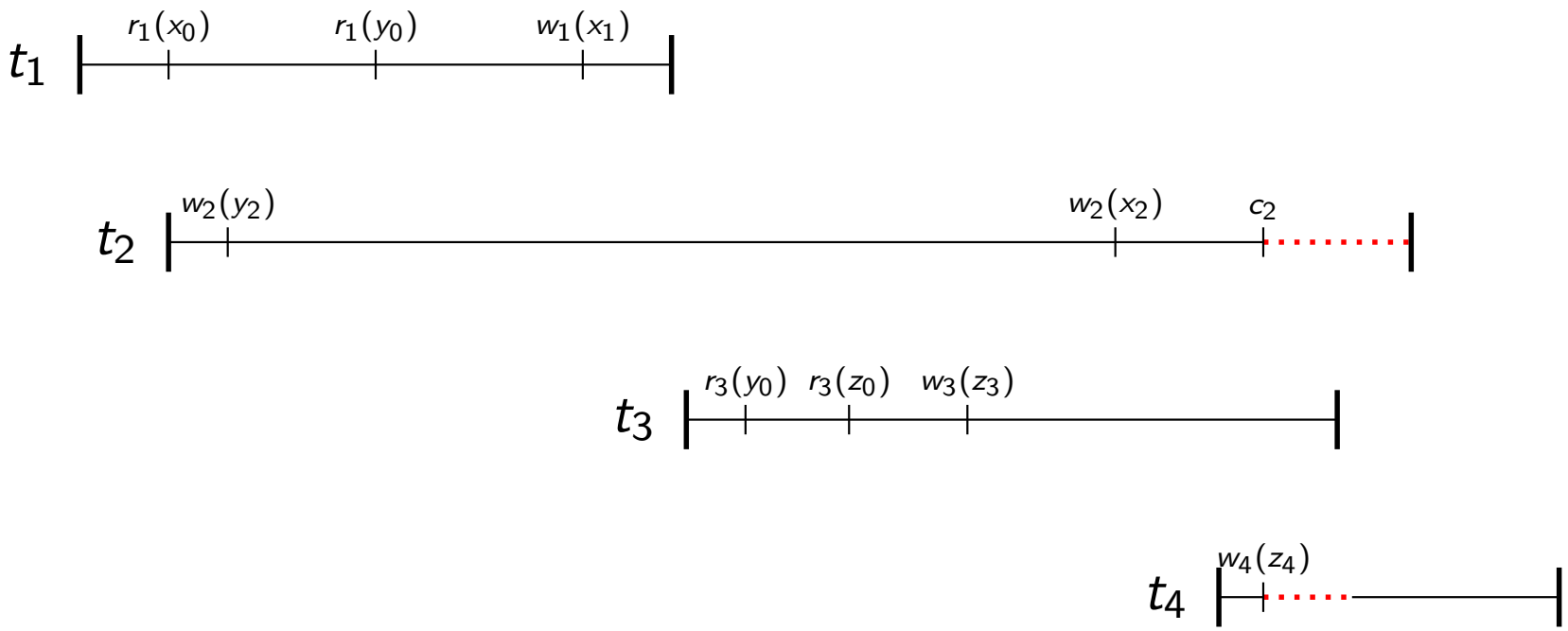
X	Y	Z
x_0	y_0 y_2	z_0



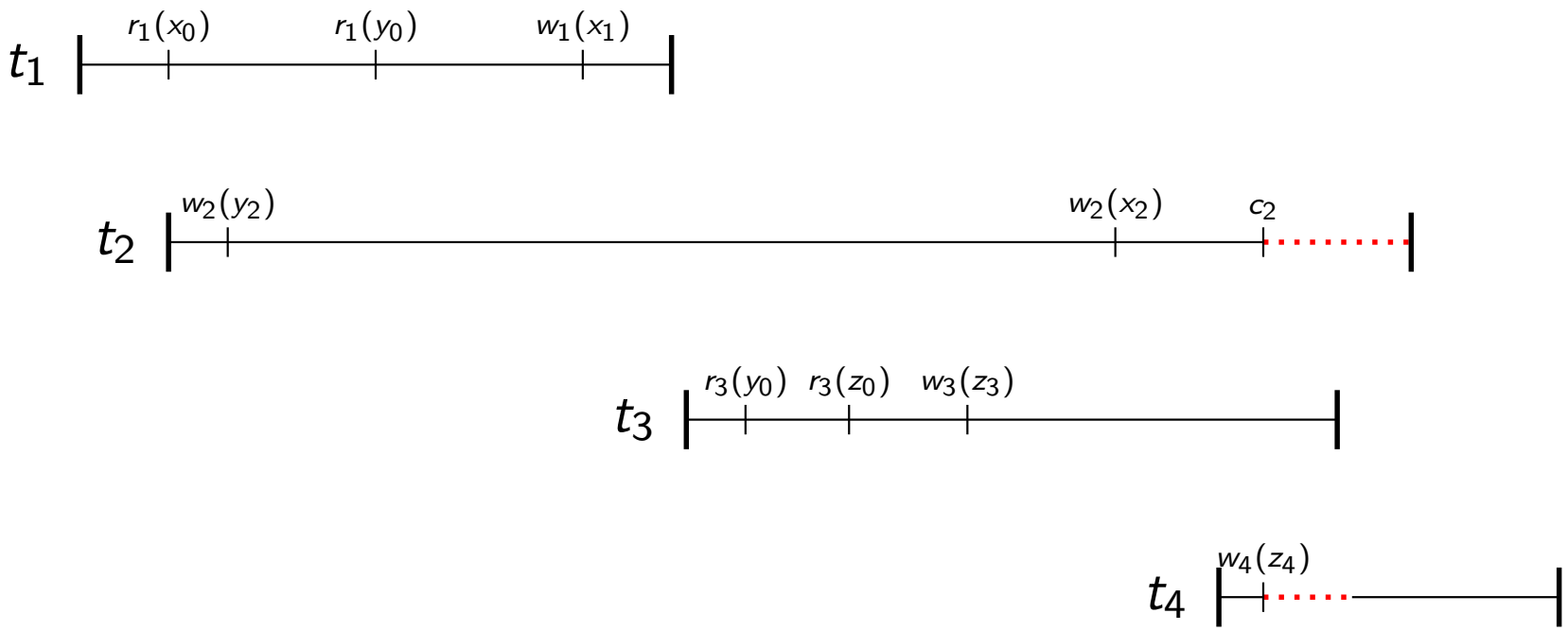
X		Y		Z
x_0	x_1	y_0	y_2	z_0



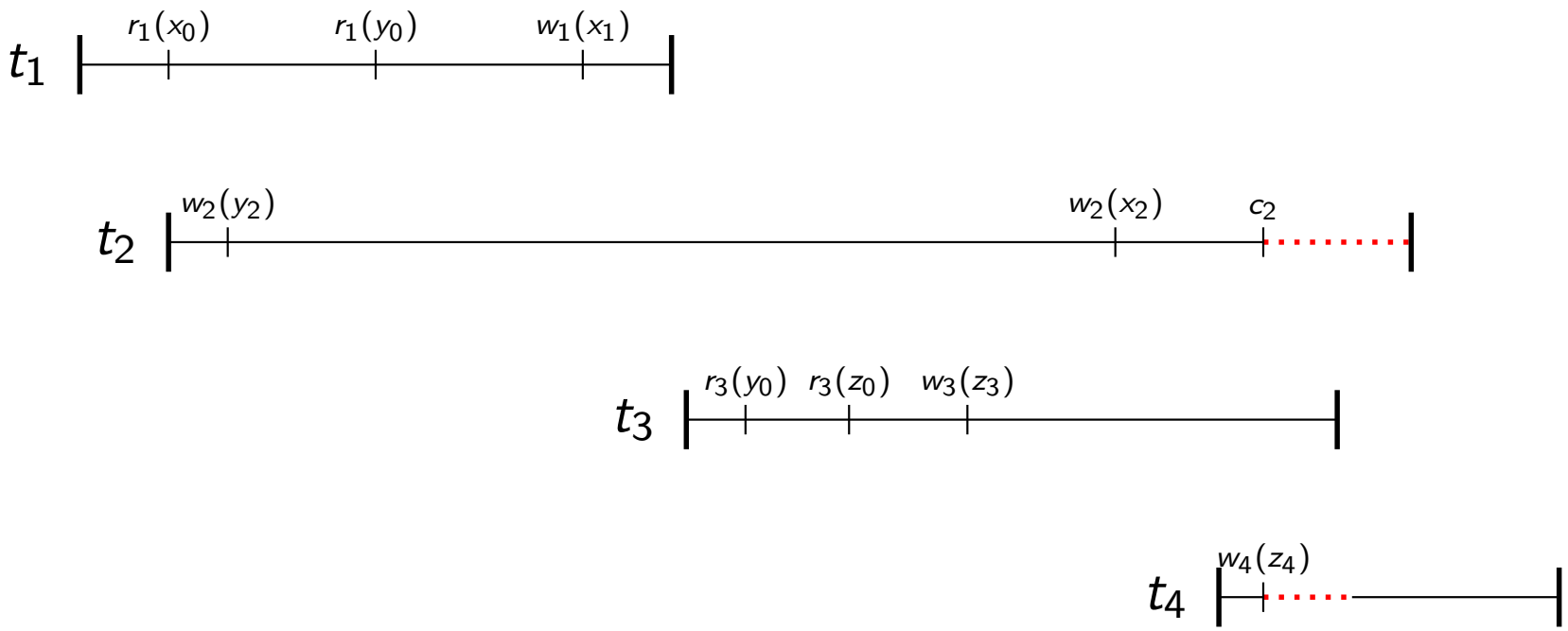
X	Y	Z
x_1	y_0 y_2	z_0



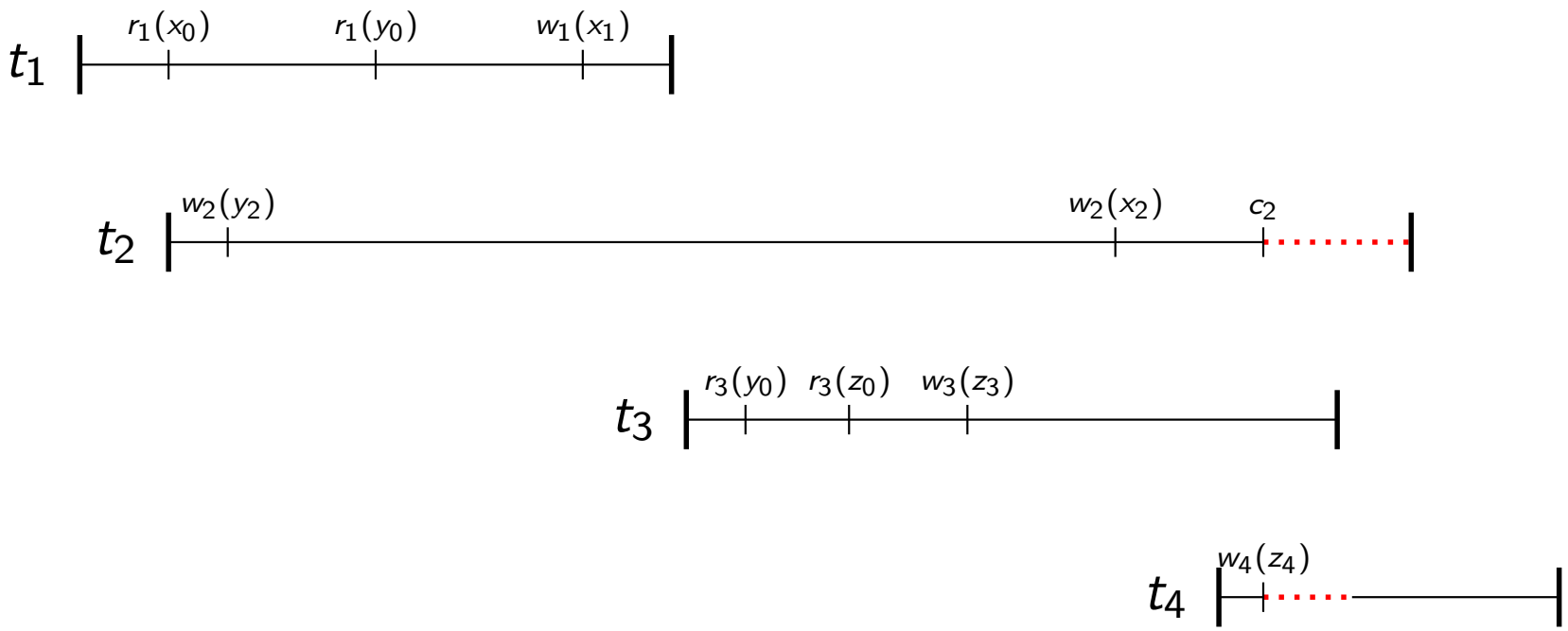
X	Y		Z	
x_1	y_0	y_2	z_0	z_3



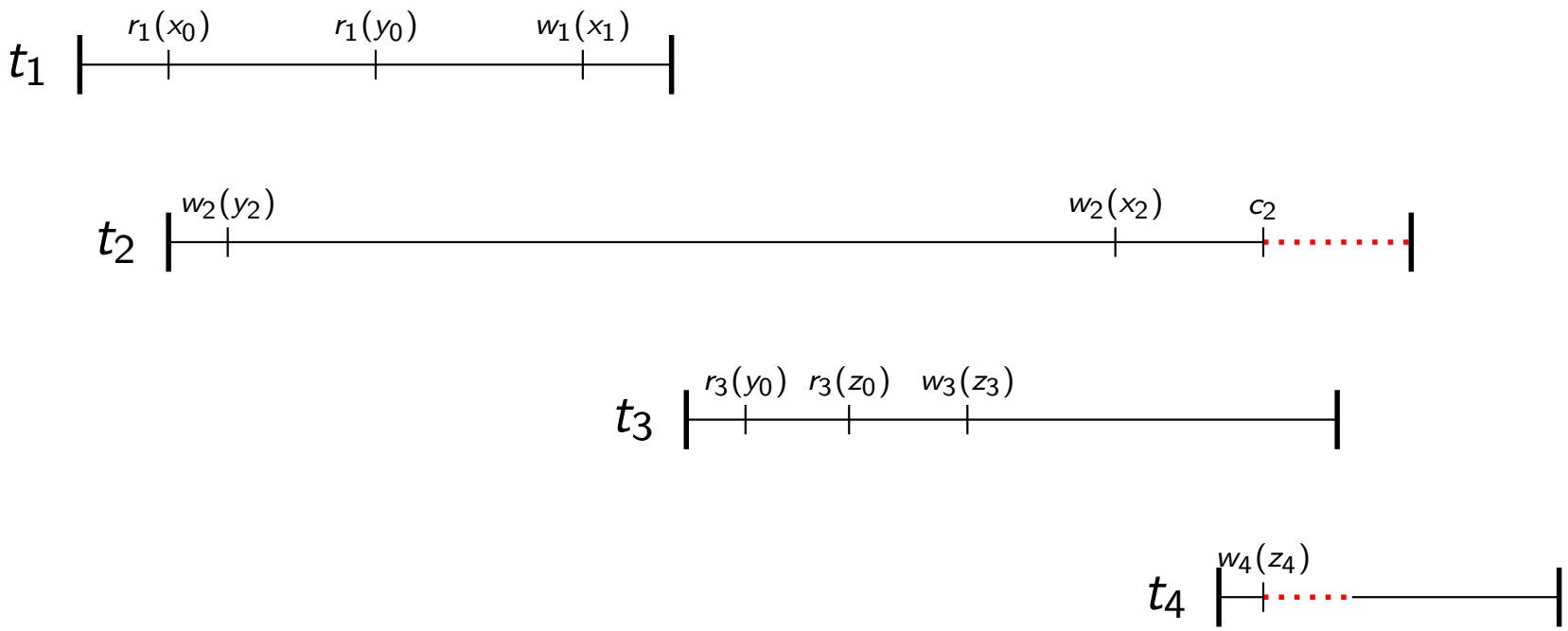
X		Y		Z	
x_1	x_2	y_0	y_2	z_0	z_3



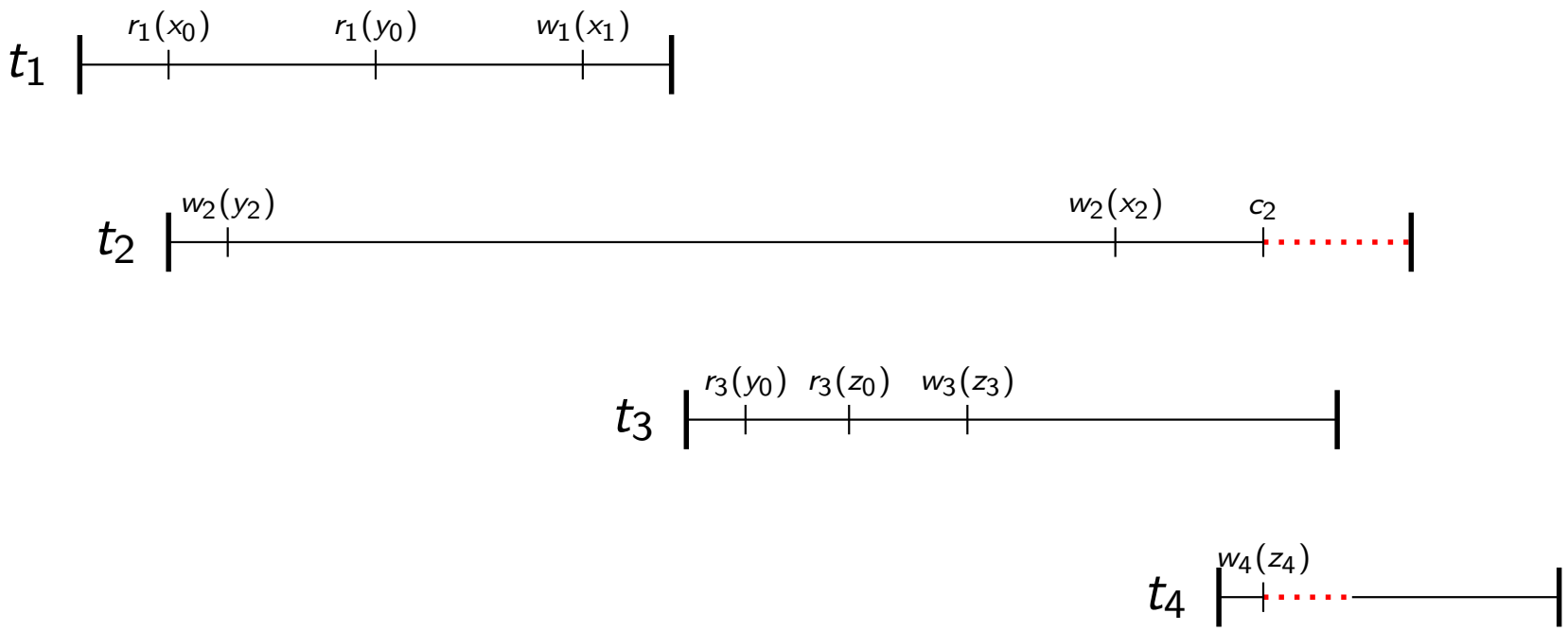
X		Y		Z
x_1	x_2	y_0	y_2	z_3



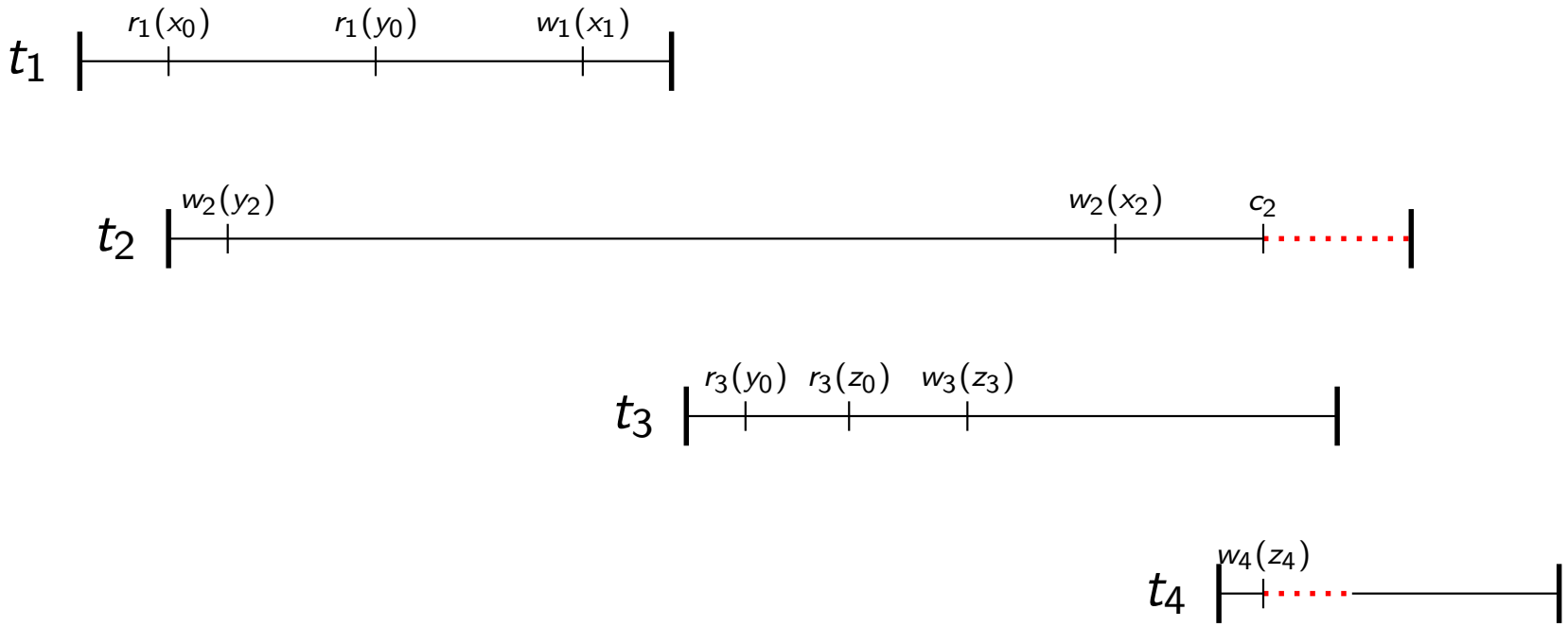
X	Y	Z
x_2	y_2	z_3



X	Y	Z	
x_2	y_2	z_3	z_4



X	Y	Z
x_2	y_2	z_4



X	Y	Z
x_2	y_2	z_4

Orden serial equivalente: $t_1; t_3; t_2; t_4$

Read Only Multiversión

ROMV

ROMV es un protocolo híbrido simple que tiene como objetivo reconciliar la simplicidad del (S)2PL convencional o de TimeStamp simple con los beneficios de rendimiento del versionado

Las transacciones de sólo lectura deben ser marcadas como tales al comenzar.

Read Only Multiversión

- **Las transacciones de actualización** están sujetas al protocolo convencional 2PL. A diferencia del entorno de monoversión convencional, cada paso de escritura **crea** una nueva versión en lugar de sobrescribir; cada versión tiene asignado el *timestamp* de su transacción que *corresponde al tiempo de commit* de la transacción.
- **Las transacciones Read Only** siguen un protocolo similar al MVTO. El timestamp asignado es el de **comienzo de la transacción** Las transacciones de solo lectura siempre acceden a la versión con el timestamp mas alto que es menor que el timestamp de la transacción,

Tanto 2PL como 2V2PL (y por lo tanto MV2PL) **no evitan** deadlocks

Bibliografía

Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.